**UNIVERSITÉ NICE SOPHIA ANTIPOLIS**

**ÉCOLE DOCTORALE STIC**
**SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION**

# T H È S E

pour obtenir le titre de

**Docteur en Sciences**

de l'Université Nice Sophia Antipolis
**Mention : INFORMATIQUE**

Présentée et soutenue par

Christian BREL

# Composition d'applications multi-modèles dirigée par la composition des interfaces graphiques

Thèse dirigée par Michel RIVEILL

préparée au sein du laboratoire I3S, Equipe RAINBOW

soutenue le 28 Juin 2013

**Jury :**

| | | | |
|---|---|---|---|
| *Rapporteurs :* | Gaëlle CALVARY | - | Professeur, INP Grenoble |
| | Jean VANDERDONCKT | - | Professeur, Université Catholique de Louvain |
| *Examinatrice :* | Sophie LEPREUX | - | Docteur, Université de Valenciennes |
| *Présidente :* | Mireille BLAY-FORNARINO | - | Professeur, Université Nice Sophia Antipolis |
| *Directeur :* | Michel RIVEILL | - | Professeur, Université Nice Sophia Antipolis |
| *Co-Encadrant :* | Philippe RENEVIER-GONIN | - | Docteur, Université Nice Sophia Antipolis |

# Résumé


## Composition d'applications multi-modèles dirigée par la composition des interfaces graphiques

Force est de constater que composer des applications existantes afin d'en réutiliser tout ou une partie est une tâche complexe. Pourtant avec l'apparition quotidienne d'applications, les éditeurs d'applications ont de plus en plus besoin d'effectuer de telles compositions pour répondre à la demande croissante des utilisateurs. Les travaux existants ne traitent généralement que d'un seul point de vue : celui du "Noyau Fonctionnel" dans le domaine du Génie Logiciel, celui des "Tâches" ou celui de l'"Interface Graphique" dans le domaine des Interactions Homme-Machine (IHM).
Cette thèse propose une nouvelle approche basée sur un modèle d'application complet (fonctionnel, tâche et interface graphique). Elle permet à un utilisateur de naviguer entre ces différents modèles pour sélectionner des ensembles cohérents pouvant être composer par substitution.
Une implémentation de cette approche a permis d'effectuer des tests utilisateurs confortant les bienfaits d'une modélisation complète.


**Mots clefs** : composition d'applications, modèle complet, interface graphique, tâches, composants logiciels, ontologies

# Abstract


## Multi-models application composition driven by user interface composition

One has to note that composing existing applications by completely or partly reusing them is a complex task. Nevertheless with the daily appearance of new available applications on any media, the application editors need to perform such compositions more and more to answer the increasing users' requests. Modeling an application for composition or just determining by which point of view on applications make this composition is not easy. Works exist, but generally deal or ensue from only a single point of view : the "Functional Core" point of view in Software Engineering field, the "Task" one or "User Interface" one in Human Computer Interaction (HCI) field.
This thesis defines a new approach based on a complete application model (functionnal, task and user interface). It enables an user to navigate between those different models in order to select consistent sets. These last ones are composable by substitution.
An implementation of this approach was used to perform user tests whose results consolidate benefits of a complete model.


**Keywords** : application composition, complete model, user interface, tasks, software components, ontologies

# Table des matières

# Première partie

# Introduction sur les besoins en composition et analyse de l'état de l'art sur la représentation et la composition d'applications

CHAPITRE 1

# Introduction

**Sommaire**



NOS travaux de recherche ont trait à la composition des applications. La composition est une des solutions pour obtenir une nouvelle application à partir d'applications déjà existantes à moindre coût et avec un gain de temps certain. Cette thèse s'inscrit dans deux domaines de recherche : le Génie Logiciel (GL) et l'Interaction Homme-Machine (IHM). Ainsi en GL, la composition de la partie métier est très développée, mais les interfaces graphiques sont à refaire. A l'inverse, les compositions en IHM traitent bien l'interface graphique ou l'activité de l'utilisateur, mais la partie métier est mise de côté. La partie métier ou l'interface graphique d'une application sont les deux parties nécessaires pour qu'elle soit opérationnelle. Nous proposons donc dans ce manuscrit de tirer parti des avancées dans les deux communautés (GL et IHM) et d'associer les deux types de compositions pour obtenir une nouvelle application opérationnelle par composition d'applications existantes.

## 1.1 Contexte, Enjeux et Problématique

### Contexte

L'explosion du nombre d'applications ou de services disponibles sur les systèmes d'exploitation bureautique ou mobile et sur le web est indéniable.

Pour accéder aux différents services, les éditeurs d'application mettent alors en place des API (*Application Programming Interface*). Ils donnent ainsi un moyen à d'autre éditeurs de pouvoir intégrer leurs services dans leurs applications et inversement, ils peuvent utiliser d'autres API pour intégrer des fonctionnalités tierces dans leurs propres applications sans avoir à reprogrammer l'existant. Ainsi les notions de service ou d'application s'entremêlent : Google Maps [1] est à la fois une application sur le web, une application sur mobile et un service utilisable par d'autres applications.

Un des besoins naissant est alors de pouvoir combiner différentes applications ou services afin de pouvoir tirer le meilleur parti de leurs fonctionnalités et de pouvoir même les combiner pour obtenir un service nouveau comblant les besoins de l'utilisateur.

C'est ainsi que le site web Doodle [2], qui permet d'organiser des réunions en réunissant les meilleurs conditions possibles (nombre de personnes maximisé en fonction des créneaux horaires proposés et de leur disponibilité), a intégré directement dans son interface la possibilité de visualiser son propre emploi du temps (Google, iCal . . . ), réduisant ainsi les allers-retours de l'utilisateur entre son application agenda et Doodle. Au niveau applications mobiles, nous pouvons aussi citer l'intégration du

1. http ://maps.google.com
2. http ://www.doodle.com

service Google Maps (visualisation de cartes, cartes routières, calcul d'itinéraire...) dans une grande variété d'applications [3] :

- «Allocine» permet de visualiser les cinémas les plus proches sur une carte sans proposer d'itinéraire,
- «Banque Populaire» permet de visualiser les distributeurs de billets de la banque et d'obtenir les itinéraires pour s'y rendre,
- «Carrefour» permet de voir l'itinéraire vers le magasin le plus proche mais seulement magasin par magasin,
- etc.

Nous constatons que l'intégration de ces services n'est pas toujours uniforme, comme l'illustre l'intégration de l'itinéraire dans les applications citées ci-dessus. Les différences peuvent s'expliquer par des utilisations différentes ou par des choix de conception différents.

La composition par intégration reste une activité complexe, puisqu'il existe des profils métiers en ingénierie informatique dédiés à cette activité : intégration, tests automatiques, sécurité, dépendances...

Il existe donc un besoin pour faciliter la composition d'application.

## Enjeux

Avec l'arrivée des *Smartphones*, téléphones portables embarquant un système d'exploitation complet et pouvant faire tourner tous types d'application, et avec l'évolution du web , il est apparu un grand nombre de magasins d'applications.

Mis à disposition de l'utilisateur, celui-ci peut alors choisir d'installer plusieurs applications répondants à ses besoins. Suivant son contexte d'utilisation, il peut utiliser plusieurs applications à la fois, basculant d'une application à l'autre pour utiliser mentalement toutes les informations en même temps. Ces allers-retours peuvent amener l'utilisateur à perdre des informations, voire à manquer d'efficacité pour les combiner.

Les éditeurs d'applications doivent dans ce contexte répondre rapidement à la demande. En effet, les utilisateurs ayant alors besoins de regrouper les différentes informations dans une seule application changeront d'application(s) afin de gagner en confort d'utilisation. Les éditeurs se doivent d'être réactifs et rapides. Éviter le développement de code spécifique à l'intégration d'une application est un moyen d'y parvenir.

**Enjeu 1 : Construire rapidement de nouvelles applications par composition de l'existant**

*L'enjeu à moyen terme est de fournir des modèles et des outils pour faciliter la composition d'applications existantes dans le but de construire de nouvelles applications opérationnelles.*

Une autre tendance émerge : laisser l'utilisateur final faire les compositions. Certains travaux comme les mashups pour le web [55, 40] laissent la main à l'utilisateur afin qu'il puisse lui-même effectuer les différentes combinaisons d'applications. Des sites web comme IGoogle [4] ou Netvibes [5] utilisent les mashups pour laisser l'utilisateur juxtaposer les différents services, ou encore les catégoriser dans différentes pages ou *dashbords*. D'autres sites comme Microsoft Popfly (service arrêté en

3. dans leur version de l'année 2012
4. http ://www.google.com/ig
5. http ://www.netvibes.com/

2009) [6], Yahoo Pipes [7], le service App Inventor [8] ou encore IFTTT.com [9] laisse un peu plus la main à l'utilisateur possédant des connaissances en informatique pour programmer des flux de données.

**Enjeu 2 : Laisser l'utilisateur final libre de composer ses propres applications par réutilisation de l'existant**

*L'enjeu à long terme, non accessible dans l'immédiat, est de fournir des outils faciles d'accès pour que l'utilisateur final puisse réaliser ses propres compositions.*

## Problématique

Bien souvent, pour développer une application, la construction de l'enchaînement des services ou fonctionnalités de l'application et la construction de son interface graphique sont deux processus séparés. Les éditeurs favorisent majoritairement le développement des fonctionnalités à mettre à disposition des utilisateurs de l'application. Une fois seulement toutes les fonctionnalités développées, ils mettent en place la construction de l'interface graphique de l'application. Durant ces processus aucune réutilisation des interfaces graphiques des applications existantes n'est généralement effectuée ou du moins cette réutilisation est limitée à l'extraction de morceaux de code à adapter à la nouvelle application.
Par conséquent, un des leviers permettant d'augmenter la rentabilité ainsi que la rapidité de développement, favorisant l'expérience déjà acquise en terme d'ergonomie et d'utilisabilité des applications, est de favoriser la réutilisation de l'existant. Cette réutilisation peut se faire grâce aux techniques de composition d'applications. Comme nous le verrons dans l'état de l'art (cf. chapitre 2), beaucoup de travaux se focalisent sur la réutilisation des applications mais à un seul niveau (en prenant en compte soit l'interface graphique de l'application, soit sa partie fonctionnelle). D'autres approches [44, 23, 46] vont utiliser des techniques de descriptions des besoins utilisateurs (les tâches), non pas pour réutiliser les applications mais pour les construire. Enfin, certains travaux comme [48, 57] traitent de la réutilisation des applications en tenant compte de plusieurs niveaux en même temps (en tenant compte du noyau fonctionnel de l'application, de son interface graphique et des liens entre ces deux parties) en prenant pour point de départ à cette réutilisation, la composition du noyau fonctionnel des applications à composer.

En se basant sur tous ces travaux, afin de répondre au premier enjeu, nous proposons d'utiliser la composition d'applications en considérant les différents niveaux de l'application : le fonctionnel, l'interface graphique et les besoins utilisateurs (exprimés à travers les tâches). Ainsi, nous prenons comme définition pour la composition d'applications, la définition suivante :

**Définition 1 : Composition d'applications**

*La composition d'applications est le moyen de combiner plusieurs morceaux d'applications existantes pour en construire une nouvelle. Cette composition réutilise tout ou une partie des applications existantes.*

Pour se faire, nous proposons de laisser le développeur effectuer cette composition d'applications par manipulation directe des morceaux des interfaces graphiques de celles-ci, tout en manipulant, sans

6. http ://www.popfly.ms/microsoft-popfly/
7. http ://pipes.yahoo.com/
8. http ://appinventor.mit.edu/
9. https ://ifttt.com/

forcément sans rendre compte, toutes les parties de l'application rattachées aux morceaux manipulés. Afin d'initier des études sur l'enjeu à long terme, nous choisissions la composition d'applications par une manipulation directe des applications à composer à travers leur partie la plus visible et la plus concrète : les interfaces graphiques.

### Cadre d'étude

La définition de cette problématique nous amène à définir le cadre de contribution de cette thèse. Nous proposons un moyen d'effectuer la composition d'applications dans le sens défini ci-dessus (cf. définition 1), c'est-à-dire permettant de combiner tout ou parties d'applications déjà existantes. Pour cela, nos travaux s'adressent tout d'abord au développeur (cf. enjeu 1) qui dirige alors la composition. Il va effectuer des choix quant aux parties d'applications à réutiliser, soit pour les inclure telle quelle dans sa nouvelle application, soit pour pouvoir les composer par substitution afin d'établir un lien entre ceux-ci. Cependant, pour pouvoir poursuivre dans le futur nos travaux en direction de l'utilisateur final (cf. enjeu 2), nous faisons l'hypothèse de ne pas avoir accès au code source des applications mais uniquement à des API. Nous pensons que les API peuvent être rendues plus facilement "accessibles" à l'utilisateur final que le code source. C'est à travers ce cadre de réutilisation de parties d'applications sans avoir accès aux sources que nous définissons notre contribution.

## 1.2 Contribution

Cette thèse a pour but de contribuer au domaine de la composition d'applications. Elle utilise des notions largement acceptées dans plusieurs autres domaines connexes comme la conception d'Interface Homme-Machine et l'Ingénierie Logicielle. La contribution de cette thèse étudie la **composition d'applications dirigée par la composition de leurs Interfaces Graphiques** (cf. partie II). Notre contribution repose sur cinq points :

1. la définition formelle d'un modèle d'application supportant la composition sur plusieurs points de vue (fonctionnalités, interface graphique et besoins),
2. des algorithmes de sélection de morceaux d'applications évoluant sur les trois mêmes points de vue de l'application (fonctionnalités, interface graphique et besoins), ces sélections étant opérationnelles,
3. une composition d'applications basée principalement sur un opérateur de substitution,
4. une mise en œuvre de ces travaux à travers un prototype qui met en exergue un processus de composition par manipulations des interfaces graphiques basées sur les trois points précédents et
5. des expérimentations utilisateur.

### Un modèle formel pour la composition d'applications dirigée par la composition des interfaces graphiques

La première contribution (cf. chapitre 3) de cette thèse est de proposer un modèle d'application qui permet la composition d'applications. Certains éléments de ce modèle font qu'il a été défini pour la composition dirigée par l'exploitation des informations disponibles à différents points de vue de l'application, que ce soit l'expression des besoins utilisateurs à travers un arbre de tâches (cf. section

2.1.2.1), ses fonctionnalités ou encore son Interface Graphique. Ce modèle s'appuie sur différents éléments qui permettent, entre autre, de proposer une composition dirigée par la manipulation des Interfaces Graphiques des applications.

### Des algorithmes de sélection sur plusieurs points de vue

La seconde contribution (cf. chapitre 4) de cette thèse est la définition et la formalisation de différentes fonctions de sélection afin d'aider le développeur dans les choix de parties d'applications à réutiliser. Pour composer en choisissant ce qui est désiré, il faut pouvoir sélectionner ce qui va être gardé et/ou substitué. Pour réaliser une composition dont le résultat soit opérationnel, il faut pouvoir garantir qu'il ne manque rien à ce qui a été choisi.

Les algorithmes de sélection exploitent des informations des autres points de vue de l'application, c'est-à-dire les informations les besoins utilisateurs à travers un arbre de tâches (cf. section 2.1.2.1), sur ses fonctionnalités ou encore au niveau de son Interface Graphique. Ils aident le meneur de la composition dans sa tâche de sélection.

### Une composition d'applications par substitution

La troisième contribution (cf. chapitre 5) de cette thèse est la proposition d'une composition basée sur un opérateur de substitutions. Disposant de différents morceaux d'applications à conserver ou à substituer pour constituer la nouvelle application, l'idée est d'établir la glu entre ces différents morceaux pour éviter la création d'une application fruit d'une juxtaposition de différentes parties d'applications existantes sans lien entre elles.

Nous proposons au meneur de la composition de pouvoir composer des applications en substituant des morceaux par un autre. Cette substitution est guidée par le rôle joué par les morceaux des applications.

### OntoCompo : un processus et un prototype pour la composition

Nous proposons un processus de composition séquentiel : sélection-substitution-placement. Ce processus est mis en œuvre dans un prototype appelé OntoCompo.

La piste exploitée dans cette thèse est que la partie la plus concrète et la plus directement manipulable d'une application est sa partie visible c'est-à-dire son Interface Graphique. C'est donc par la manipulation de cette interface graphique, pour la sélection puis pour effectuer les substitutions, que nous laissons le développeur dirigé la composition. C'est ainsi que les éléments d'interfaces graphiques redondants (car provenant de plusieurs applications) pourront être substitués afin de n'en garder qu'un seul ou de les remplacer par un autre.

### Retours d'expérimentations

A partir du prototype réalisé, nous avons mené une série d'expérimentations avec des informaticiens-développeurs. Le but de celle-ci est tout d'abord d'éprouver le bien fondé de nos modèles, puis d'obtenir des informations quant aux limites possibles du processus de composition basé uniquement sur la manipulation des interfaces graphiques. Enfin elles nous ont permis de définir des pistes à explorer pour aller vers une composition menée par l'utilisateur final.

## 1.3 Structure de la thèse

Ce manuscrit est construit autour de quatre parties principales.

La première partie est articulée autour de deux chapitres, le premier est cette introduction au contexte, enjeux et contributions de cette thèse. Le second analyse les travaux existants dans un état de l'art articulé autour de deux axes : le premier sur les modèles pour définir une application, que ce soit sa partie fonctionnelle, sa partie graphique et ses tâches, et le second sur les travaux traitants de la composition d'applications.

La seconde partie est articulée autour de trois chapitres, mettant en évidence notre contribution. Le premier chapitre traite du modèle que nous proposons pour définir une application et les différents points de vue de celle-ci. Le second chapitre traite des algorithmes mis en place pour aider le développeur dans ces choix de morceaux d'application à conserver et/ou à substituer. Enfin le dernier chapitre décrit la composition par substitution que nous proposons afin de lier les morceaux d'applications en une nouvelle application qui soit opérationnelle.

La troisième partie est articulée autour de deux chapitres. Le premier chapitre décrit la mise en œuvre de notre contribution à travers le développement d'un prototype puis le second chapitre décrit les résultats des expérimentations menées auprès d'utilisateur sur ce prototype.

Enfin la dernière partie est constituée d'un chapitre faisant le bilan de ce manuscrit avec un retour sur nos objectifs et ouvrant cette thèse vers de nouvelles perspectives de recherche.

## 1.4 Etude de cas

Cette section décrit un exemple de composition «fil rouge» qui nous permet d'illustrer les différents modèles et algorithmes présents dans la contribution de cette thèse. Elle utilise la composition de deux applications à la fois assez simples dans leur fonctionnement et assez pertinentes pour illustrer nos apports.

### Recherche de cinémas les plus proches et séances prévues par cinéma

La première application est une application de recherche de cinémas (cf. figure 1.1). Elle permet de découvrir une liste de cinémas les plus proches d'une adresse donnée. La première liste, située à gauche de l'interface graphique, affiche un cinéma par ligne avec son nom, son adresse accompagnée du code postal et de la ville associée, le nombre de salles disponibles. Lors de la sélection d'un cinéma dans la liste, une seconde liste, située à droite sur l'interface graphique, donne la liste des séances de films prévues dans le cinéma sélectionné.

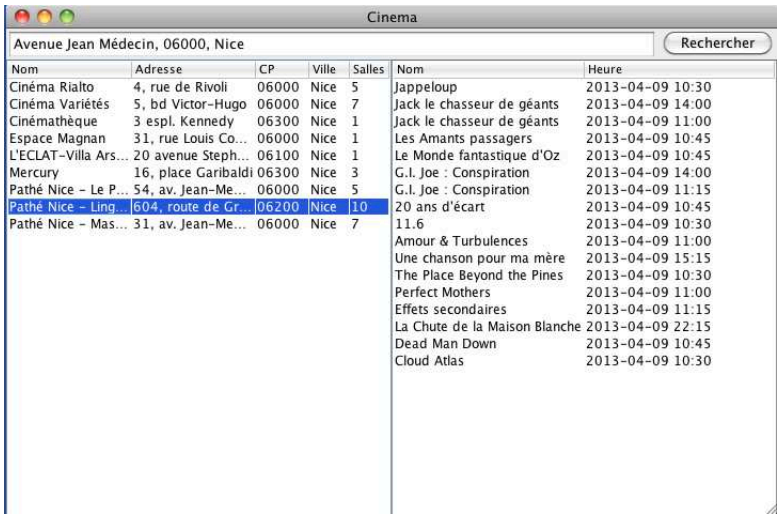



FIGURE 1.1 – Interface graphique de l'application "Cinema" de recherche de cinémas les plus proches d'une adresse donnée.

## Calcul d'itinéraires

La seconde application est une application de calcul d'itinéraire (cf. figure 1.2). Elle permet à partir de deux adresses de calculer le trajet et de l'afficher sur une carte routière. Elle affiche aussi les principales intersections de l'itinéraire. Il est possible d'effectuer un zoom sur la carte à l'aide du "slider" présent dans l'interface graphique ou en agissant directement sur la carte.

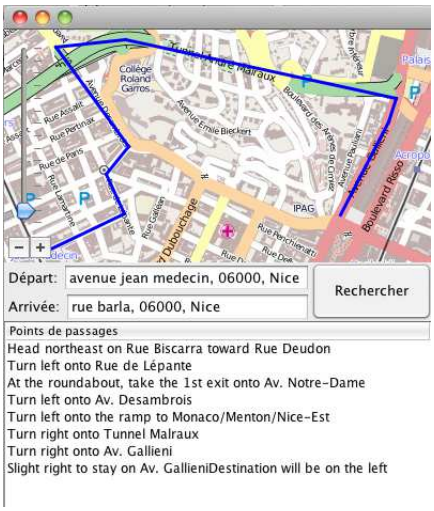



FIGURE 1.2 – Interface graphique de l'application de calcul d'itinéraire "Maps".

## Composition d'applications "Cinema" et "Maps"

Nous souhaitons après avoir sélectionné un cinéma pouvoir afficher l'itinéraire pour nous y rendre (cf. figure 1.3). Cette nouvelle application, obtenue par composition, permet dans un premier temps d'obtenir la liste des cinémas les plus proches de l'adresse renseignée (fonctionnalité qui provient

de l'application "Cinema"). A cela, la composition ajoute la fonctionnalité d'affichage de l'itinéraire (provenant de l'application "Maps") entre l'adresse renseignée et le cinéma sélectionné dans la liste. Pour obtenir une application opérationnelle, l'application résultante doit fournir l'adresse renseignée comme adresse de départ pour le calcul de l'itinéraire et l'adresse du cinéma comme adresse d'arrivée. Enfin, le déclenchement de l'affichage de l'itinéraire doit s'effectuer lors de la sélection du cinéma dans la liste.

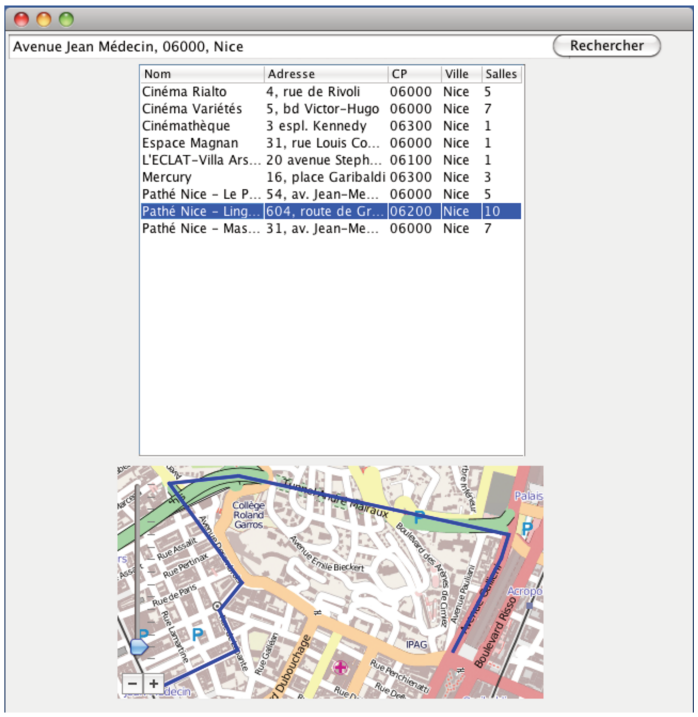



FIGURE 1.3 – Interface graphique de l'application obtenue par composition entre "Cinema" et "Maps".

CHAPITRE 2

# Etat de l'art

**Sommaire**



NOUS présentons dans ce chapitre les travaux relatifs à la composition d'applications. Dans un premier temps, nous décrivons les modèles permettant de représenter une application. Nous commençons par les représentations de la partie qui effectue les calculs et traitements de l'application qui est son Noyau Fonctionnel. Puis nous présentons les travaux relatifs à l'interface graphique. Certains travaux autour de la construction d'une interface graphique mettent en avant l'utilisation d'arbre de tâches dont nous décrivons certaines représentations. L'objet de la seconde partie de cet état de l'art est la composition d'applications où nous l'étudions suivant les différents points d'entrée utilisés pour effectuer cette composition à savoir, les compositions dirigées par la manipulation du Noyau fonctionnel de l'application, les compositions dirigées par la manipulation des tâches et les compositions dirigées par la manipulation des interfaces graphiques. Nous complétons ensuite cette partie de l'état de l'art sur la composition sur le Web.

## 2.1 Représentation d'une application

Afin de pouvoir composer des applications, il est tout d'abord nécessaire d'en définir une représentation. Pour cela, nous pouvons nous appuyer sur les différentes architectures d'applications comme MVC [15], PAC [4] ou encore Arch [1]. La plupart décompose l'application suivant différentes préoccupations propre à chaque approche mais une claire distinction est faite entre la partie graphique de l'application (son IHM - Interface Homme-Machine) c'est-à-dire en contact avec l'utilisateur final, et sa partie fonctionnelle (son NF - Noyau Fonctionnel). Ces deux parties peuvent être distinguées en sous-parties suivant les approches, cependant les différentes architectures décrivent la manière dont ces deux parties de l'application sont connectées. Nous étudions donc par la suite les différentes représentations de ces deux parties de l'application.

### 2.1.1 Représentation du NF de l'application

Les possibles représentations pour le NF d'une application sont nombreuses mais nous nous focalisons ici sur les représentations donnant accès à une composition au sens où nous l'avons défini

dans la section 1.1, c'est-à-dire qui permettent une réutilisation de tout ou d'une partie de l'application. Deux types de représentations prédominent alors : l'approche utilisant des services et l'approche basée sur des composants.

#### 2.1.1.1 Architecture Orientée Service

L'Architecture Orientée Service (*Service Oriented Architecture - SOA* [5]) a pour noyau l'utilisation de parties logicielles mettant à disposition une fonctionnalité ou partie de fonctionnalité permettant de remplir une tâche particulière, de "rendre" un service. Une définition d'une Architecture Orientée Service donnée par l'OASIS (Organization for the Advancement of Structured Information Standards) est la suivante :

**Définition 2 : Architecture Orientée Service**

*Service Oriented Architecture is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations. [12]*

Les services implémentent une ou plusieurs interfaces correspondant aux différentes actions spécifiques qu'ils peuvent réaliser. Ils ont un couplage faible entre-eux car ils sont interconnectés par des standards qui assurent une réduction des dépendances. Ces standards sont souvent basés sur des échanges de documents en langage XML. Une autre caractéristique des services est qu'ils doivent être identifiables et localisables, c'est-à-dire qu'avant de pouvoir appeler un service, il faut pouvoir le trouver. C'est ainsi que les systèmes d'informations reposant sur une architecture basée services mettent en place un annuaire qui permet d'enregistrer les autres services (soit de manière extérieure, soit de manière autonome - le service lui-même va s'enregistrer dans cet annuaire). Le référencement de services le plus utilisé est UDDI (Universal Description Discovery and Integration) [13]. La plupart du temps, l'architecture basée services met en relation les différents services qui la compose à travers un bus logiciel qui va donc servir d'intermédiaire entre le consommateur du service et le producteur du service. L'annuaire et le bus de services contribuent donc dans une architecture orientée services au couplage faible entre les services.

Une déclinaison "sur internet" des services sont les Web Services. Il existe deux types de Web Services. Les Web Services (qui respectent les spécifications WS-* [1] ) utilisent WSDL (Web Services Description Language) [17] comme langage XML de description et SOAP (Simple Object Access Protocol) [19] comme protocole de communication utilisant lui même HTTP comme protocole de transport. Les Web Services REST (REpresentational State Transfer) [6] quant à eux exposent leurs fonctionnalités comme des ressources identifiables en utilisant les URI (Uniform Resource Identifier) et sont aussi accessibles via le protocole de transport HTTP. Pour automatiser l'utilisation des Web Services et notamment leur découverte, leur invocation, leur contrôle, OWL-S (Ontology Web Language for Services) [18] permet une description sémantique des services (vient en complément à la description WSDL) découpée en trois parties, le profile du service (*service profile*) - qui décrit ce que fait le service et est plutôt destiné à être lu par des humains -, le modèle du service (*service model*) -

1. http ://en.wikipedia.org/wiki/List_of_web_service_specifications

qui décrit comment fonctionne le service (entrées, sorties, pré-conditions, résultats ...) -, et les bases du service (*service grounding*) - qui détaillent comment interagir avec le service, les protocoles de communication, les formats de message etc ...

#### 2.1.1.2 Programmation par composants

Une deuxième approche pour construire une application et sa partie fonctionnelle est l'utilisation de composants (*Component-Based Software Engineering - CBSE* [7, 16]). Cette approche propose l'utilisation de "parties applicatives" qui peuvent être réutilisées et assemblées pour former une application. Une définition largement admise d'un composant est celle donnée par Szyperski :

**Définition 3 : Composant**
*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. [16]*

Il existe différents modèles pour les composants [11] mais leur succès est basé sur quatre principes largement acceptés [11] :
- l'utilisation d'entités logicielles pré-construites et pré-existantes (on parle alors de Composants sur Étagère - *Commercial Off-The-Shelf, COTS*),
- les éditeurs/développeurs de composants ne sont pas nécessairement les mêmes personnes que les développeurs qui les intègrent dans les applications,
- un composant peut être utilisé plus d'une fois c'est-à-dire que contrairement aux services où l'instance d'un service est unique, il peut y avoir plusieurs instances d'un composant et
- un composant est "composable" c'est-à-dire qu'il peut prendre place dans un composant dit composite (un composant utilisant d'autres composants pour fournir une fonctionnalité de plus haut niveau) et les composants composites peuvent eux-même être composables.

Ainsi, cette approche permet de capitaliser du code dans des entités logicielles dites boites noires qui sont alors réutilisables et dont seules les interfaces logicielles d'échange sont connues. Ces interfaces sont l'une des caractéristiques principales des composants puisque c'est à travers celles-ci que les composants sont interconnectés. Ces interfaces peuvent être de deux types, elles peuvent être fournies ou requises. Les interfaces fournies décrivent, à la manière des services, ce que propose le composant en terme de fonctionnalités. Les interfaces requises quant à elles décrivent les dépendances entre composants, c'est-à-dire ce qu'il est nécessaire de fournir au composant pour son bon fonctionnement. A la différence des services qui ont une existence en soi, les composants prennent leur existences dans un assemblage de composants pour former une application et sont donc connectés à travers leurs interfaces respectives, une interface fournie étant connectée à une interface requise. Comme exemples de modèles de composants nous citerons Fractal [3], SCA [2, 14] ou SLCA [8].

#### 2.1.1.3 Autres représentations orientées composition du Noyau Fonctionnel

D'autres approches comme la Programmation Orientée Aspect (*Aspect Oriented Programming - AOP* [10]) ou les modèles de variabilités (*Features Model*) [9] permettent aussi une modularisation des applications et d'obtenir un certain découpage du code de l'application.

Dans le cas des aspects, ils sont insérés dans un code ou un *bytecode* déjà existant afin de rajouter des fonctionnalités non présentes dans le code de départ comme de la sécurité, de la persistance, du

monitoring etc. Il faut donc avoir accès aux codes sources de l'application ou être capable d'analyser le *bytecode*, ce qui est contraire à notre cadre d'étude défini en introduction (cf. 1).

Les Features Models quant à eux permettent de pouvoir garantir une certaine cohérence entre les choix de fonctionnalités effectués, puisque pour chacune d'entre elle, nous pouvons exprimer un ensemble de contraintes sur lesquelles un raisonnement peut être tenu pour déterminer une configuration (un ensemble de features *i.e.*, de caractéristiques sélectionnées) correcte.

#### 2.1.1.4 Synthèse sur la représentation du Noyau Fonctionnel d'une application

Nous nous sommes focalisés dans cette partie sur deux types d'architecture permettant de définir le Noyau Fonctionnel d'une application. Ces deux types d'architecture sont en fait très proches puisque dans un cas, l'application va être construite autour d'un ensemble de services et dans l'autre cas, autour d'un ensemble de composants, dont chacun va dans un sens rendre un service. Il est même possible d'encapsuler des services dans des composants pour les utiliser dans des assemblages de composants [8]. L'inverse est également possible.

Ces approches mettent en place un couplage faible entre les éléments qui permettent une grande réutilisation de ceux-ci. La différence entre les deux approches réside dans le fait que pour les services l'orchestration est explicite et nous connaissons exactement le cheminements des données et leurs transformations. A l'inverse, dans l'approche composants, ceux-ci peuvent être vus comme des boîtes noires où le cheminement des données n'est pas à priori pas connu.

### 2.1.2 Représentation de l'IHM de l'application

La communauté IHM préconise de développer des interfaces à différents niveaux d'abstraction de manière indépendante afin de produire des IHM de qualité et exploitables dans divers contextes (différents systèmes d'exploitation, différents langages de programmation, différents supports tels que les Pcs, les téléphones, etc). Ces niveaux ont été définis par le projet Cameleon dans un framework de référence, le *Cameleon Reference Framework* [24]. A partir de n'importe lequel de ces niveaux, le développeur peut obtenir différentes variantes d'une même IHM selon le contexte d'utilisation. Quatre niveaux d'abstraction sont identifiés :

- les objectifs des utilisateurs, les procédures pour les atteindre et concepts du domaine qui décrivent l'IHM selon une perspective besoin des utilisateurs, sans préjuger, d'une quelconque représentation. Les concepts du domaine sont les entités manipulées par les tâches présentées dans la section suivante,
- l'interface abstraite qui structure l'IHM en espaces de dialogue (une page web par exemple) et fixe la navigation entre ces espaces. A ce niveau, on reste indépendant de tout contexte d'utilisation,
- l'interface concrète qui affine l'IHM en introduisant les choix d'interacteurs (boutons, menus, etc.) en fonction d'un contexte d'utilisation donné (l'action "valider" peut être associée à un clic sur un bouton dans une IHM pour un poste de travail ou à une commande vocale dans le cadre d'une IHM sur téléphone portable dans un contexte "mains libres" par exemple) et
- l'interface finale, qui correspond à implémenter l'interface concrète dans un langage donné (Java SWING, Adobe Flex, HTML, etc.) et un système d'exploitation spécifique. C'est à travers ce dernier niveau de description qu'il est possible à l'utilisateur de manipuler l'interface de l'application.

Différents langages ont été définies pour décrire ces différents niveaux d'abstraction des IHM. Nous présentons par la suite certains langages pour décrire les tâches, l'interface abstraite ou l'interface concrète. L'interface finale quant à elle revient au code s'exécutant directement sur la plate-forme cible donc tout langage de développement permettant de construite une interface graphique est valable comme langage de description d'une interface finale.

#### 2.1.2.1 Langages de descriptions des tâches

Le domaine de l'Analyse des Tâches (*Task Analysis*) permet l'étude de la manière dont est réalisé un objectif par un utilisateur, avec une description des activités manuelles ou intellectuelles pour la réaliser, le temps de réalisation, sa fréquence, sa complexité... Nous pouvons trouver une définition d'une tâche dans [34] :

**Définition 4 : Tâche**
*A task defines how the user can reach a goal in a specific application domain. The goal is a desired modification of the state of a system or a query to it. [34]*

Un modèle de tâches décrit l'enchaînement des activités que doit réaliser un utilisateur pour atteindre son but. Les enchaînements sont définies par des opérateurs variant d'un modèle à un autre mais la plupart sont des opérateurs de choix, de séquentialités ou de parallélismes. Plusieurs modèles de tâches ont été proposées comme HTA (*Hierarchical Task Analysis*) [22, 21], UAN (User Action Notation) [27], GOMS (*Goals, Operators, Methods and Selection rules*) [25, 29], ou encore CTT (ConcurTaskTrees) [34, 33]. Certains proposent de regrouper les propriétés récurrentes de ces modèles pour former des modèles uniformisés comme [30] pour une application Ingénierie Dirigée par les Modèles - IDM (*Model-Driven Engineering - MDE*) ou encore pour une application sémantique avec les ontologies décrites dans [37] et [38].

Les principales caractéristiques de ces modèles sont les suivantes. Une tâche peut-être simple (*i.e.*, non décomposée) ou complexe. Les tâches complexes sont composées de sous-tâches ce qui amènent les modèles de tâches à être hiérarchiques. Une tâche complexe est qualifiée de "tâche de plus haut niveau" par rapport aux tâches qui la composent. Avec cette hiérarchie, il est nécessaire de décrire l'ordre dans lequel les tâches doivent être exécutées c'est-à-dire de décrire le *task flow*. Cela s'effectue à l'aide d'opérateurs temporels comme ceux utilisés dans CTT qui sont les opérateurs LOTOS [28] (cf. figure 2.1).

Dans les systèmes interactifs, les tâches peuvent-être de plusieurs types :

- les tâches utilisateurs (*User Tasks*) sont des tâches qui ne nécessitent pas d'interactions avec le système et qui définissent une activité réalisée par l'utilisateur qu'elle soit physique ou cognitive
- les tâches systèmes (*System Tasks*) sont des tâches qui sont réalisées par le système sans intéraction avec l'utilisateur (par exemple, "calculer l'itinéraire entre la position A et la position B")
- les tâches d'intéractions (*Interaction Tasks*) sont des tâches qui permettent de décrire une intéraction entre l'utilisateur et le système, c'est-à-dire les tâches qui donne le moyen à l'utilisateur de dialoguer avec le système (par exemple, de l'utilisateur vers le système, entrer une adresse de départ ou autre exemple, du système vers l'utilisateur, pour afficher un itinéraire sur une carte)
- les tâches abstraites sont des tâches complexes qui sont composées d'au moins deux sous-tâches qui peuvent être de n'importe quel type.

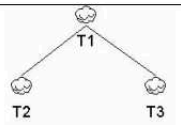
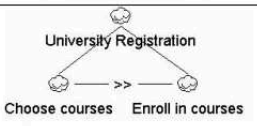
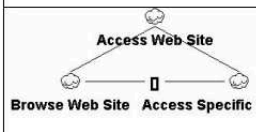
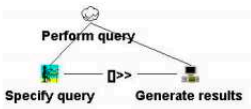
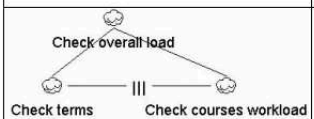
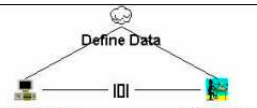
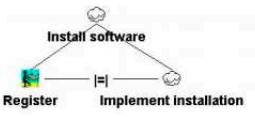
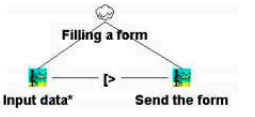
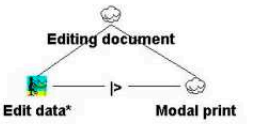
| | |
|---|---|
|  | **Hierarchy**<br>Tasks at same level represent different options or different tasks at the same abstraction level that have to be performed.<br>Read levels as "In order to do T1, I need to do T2 and T3", or "In order to do T1, I need to do T2 or T3" |
|  | **Enabling**<br>Specifies second task cannot begin until first task performed.<br>Example: I cannot enroll at university before I have chosen which courses to take. |
|  | **Choice**<br>Specifies two tasks enabled, then once one has started the other one is no longer enabled.<br>Example: When accessing a web site it is possible either to browse it or to access some detailed information. |
|  | **Enabling with information passing**<br>Specifies second task cannot be performed until first task is performed, and that information produced in first task is used as input for the second one.<br>Example: The system generates results only after that the user specifies a query and the results will depend on the query specified. |
|  | **Concurrent tasks**<br>Tasks can be performed in any order, or at same time, including the possibility of starting a task before the other one has been completed.<br>Example: In order to check the load of a set of courses, I need to consider what terms they fall in and to consider how much work each course represents |
|  | **Concurrent Communicating Tasks**<br>Tasks that can exchange information while performed concurrently<br>Example: An application where the system displays a calendar where it is highlighted the data that is entered in the meantime by the user. |
|  | **Task independence**<br>Tasks can be performed in any order, but when one starts then it has to finish before the other one can start.<br>Example: When people install new software they can start by either registering or implementing the installation but if they start one task they have to finish it before moving to the other one. |
|  | **Disabling**<br>The first task (usually an iterative task) is completely interrupted by the second task.<br>Example: A user can iteratively input data in a form until the form is sent. |
|  | **Suspende-Resume**<br>First task can be interrupted by the second one. When the second terminates then the first one can be reactivated from the state reached before<br>Example: Editing some data and then enabling the possibility of printing them in an environment where when printing is performed then it is no possible to edit. |

FIGURE 2.1 – Opérateurs temporels LOTOS utilisés dans CTT tiré de [33].

Pour les systèmes interactifs, une modélisation des concepts du domaine manipulés dans les tâches est ajoutée au modèle de tâches. Il décrit ces concepts sous la forme d'objets matériels (physiquement tangible) ou mentalement présents dans l'esprit de l'utilisateur du système. Dans le cadre des travaux qui génèrent une interface à partir de la description des tâches, certains objets sont fortement rattachés à un élément graphique de l'interface finale. L'intégration du modèle d'objets dans le modèle de tâches se fait à différents niveaux suivant les modèles. Soit les objets sont intégrés comme éléments d'entrée/sortie d'une tâche et donc la sortie d'une tâche sert d'entrée à la suivante, ce qui permet le

passage d'information entre les tâches. Soit les objets voient leur propriétés modifiées par les tâches et il est défini différents états (initial et final) pour les valeurs de ces propriétés. Ainsi, le passage d'information entre tâches se fait indirectement à travers la modification des propriétés des objets.

La modélisation de l'arbre de tâches d'une application permet de décrire les besoins utilisateurs et les actions permises via l'application. Les tâches regroupent ainsi les interactions entre l'interface graphique de l'application et les actions système de sa partie fonctionnelle selon l'activité de l'utilisateur.

Cette modélisation permet une vérification "manuelle" de la conformité de la conception logicielle et de la conception de l'interface graphique. Des implémentations directes de celle-ci existent aussi en tant que "contrôleur de dialogue" par exemple à travers sa mise en œuvre dans des réseaux de Petri [71].

#### 2.1.2.2 Langages de description d'interfaces abstraites ou concrètes

Beaucoup de langages de Description d'Interface Utilisateurs suivent la structure décrite par le Cameleon Reference Framework (CRF) [24]. UIML [20], SunML [36], UsiXML [31] ou MARIA [35] font partis de ces langages (*User Interface Description Languages - UIDLs*).

SunML est un langage XML basé sur un nombre restreint de concepts afin de décrire l'interface graphique d'une application. Il permet de décrire principalement la structure de l'interface de manière abstraite. Enfin, il introduit la notion de réutilisation d'une partie d'interface par le fait de pouvoir faire référence à un morceau d'interface décrit dans un autre fichier SunML.

UIML est aussi un langage XML permettant de décrire l'interface graphique à un niveau abstrait et concret avec l'objectif de décrire des interfaces multi-dispositifs. Le langage est structuré autour de cinq balises XML qui sont *description*, *structure*, *data*, *style* et *events*. Les balises *description*, *structure* et *data* permettent de décrire l'interface d'une manière abstraite, avec la liste des éléments constituants l'interface, sa structure mais aussi les informations à afficher dans les éléments. Les balises *style* et *events* permettent de décrire l'interface d'une manière plus concrète et elles possèdent des propriétés dont les valeurs font référence à des éléments de la plate-forme d'exécution de l'interface.

MARIA et UsiXML implémentent les différents modèles présents dans le *CRF* pour la construction d'une interface graphique. UsiXML en est l'implémentation stricte dans le sens où il est possible de débuter la description de l'interface par n'importe quel modèle (tâches, interface abstraite, interface concrète, interface finale) puis de passer d'un modèle à l'autre par transformations de modèle. Des liens entre modèles sont aussi présents puisque le *MappingModel* permet de lier des modèles ou éléments de modèles entre eux. UsiXML permet notamment aux designer d'interface de créer des interfaces multi-modales voire même pouvant s'exécuter sur plusieurs dispositifs. Le but principal d'UsiXML est de pouvoir construire son interface à partir d'un des niveaux d'abstraction présents dans le *CRF* puis, de maintenir à jour les différents modèles à travers différentes transformations de modèles, remontrer en abstraction (transformation d'*Abstraction*)) ou au contraire redescendre vers des niveaux plus concrets (transformation de *Réification*) tout cela en pouvant changer de contexte d'utilisation (transformation de *Translation*). Un outil UsiComp [26] permet de mettre en œuvre cette approche à base de transformations de modèles. Il permet aussi de pouvoir effectuer ces transformations au runtime. Enfin, l'outil permet l'interaction avec d'autres outils disponibles qui vont permettre de générer automatiquement un arbre de tâches ou encore une interface concrète. Le résultat de ces

outils peut alors être adapté avec une transformation de modèle afin de correspondre aux modèles utilisés dans UsiComp et ainsi bénéficier de ses fonctionnalités pour générer l'interface graphique de l'application.

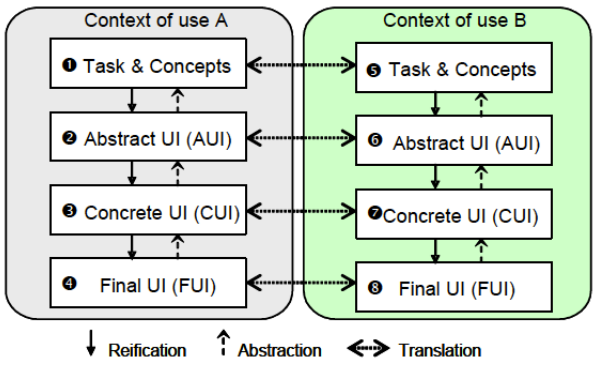



FIGURE 2.2 – Niveaux d'abstraction du CRF avec leurs transformations tiré de [31].

MARIA, évolution de TERESA [32], est plus destiné à des applications basées sur des web services. Ces travaux décrivent les interfaces à travers tous les modèles du *CRF* avec un ajout pour gérer les applications *multi-touch* et les technologies permettant une mise à jour d'une partie de l'interface de manière asynchrone (AJAX par exemple). La principale évolution entre TERESA et MARIA est l'externalisation des transformations de modèles. Ces changements permettent de prendre en compte l'évolution des applications qui de nos jours sont de plus en plus basés sur l'appel de services extérieurs (web services, services REST etc.). A partir de la description de l'arbre de tâches (cf. section précédente), il est possible de décrire l'interface abstraite qui à son tour va permettre de décrire les interfaces concrètes mettant en jeu les différentes modalités d'interactions présentes dans l'interface mais aussi de pouvoir par la suite se projeter sur différentes plates-formes cibles.

#### 2.1.2.3 Synthèse sur la représentation de l'interface graphique d'une application

Nous avons vu que les différentes approches pour la représentation de l'interface graphique d'une application s'articulent principalement autour d'un framework de référence (le *CRF*) qui décrit quatre niveaux d'abstraction pour une interface. Le niveau le plus abstrait est celui de la description des tâches de l'application et il peut se reposer sur plusieurs langages pour le décrire. Ces derniers mettent en place une représentation hiérarchique des tâches dont certaines vont pouvoir être détaillées par des sous-tâches à réaliser. Les tâches dans cet arbre vont alors être organisées par des relations temporelles entre celles-ci.

Les niveaux d'interface abstraite, concrète puis finale décrivent les interfaces "graphiques". L'interface abstraite va être indépendante de tout contexte d'utilisation et permet de définir les espaces de dialogue de l'interface. Cette description fait ressortir la structure hiérarchique de l'interface. L'interface concrète permet de définir les interacteurs en fonction d'un contexte d'utilisation. Enfin l'interface finale va correspondre à l'interface implémentée dans un langage de programmation.

## 2.2 Composition d'Applications

A partir des différentes descriptions pour une application présentées dans la section précédente, plusieurs sortes de composition d'application sont étudiées. A partir des trois points de vue pour décrire une application qui sont (i) la description du NF de l'application, (ii) l'arbre de tâches et (iii) la description de l'interface graphique de l'application. Chaque point de vue constitue alors un point d'entrée pour pouvoir effectuer la composition. Un point d'entrée va alors correspondre à la manière dont la composition peut être construite. Chaque point d'entrée aborde un problème spécifique de la composition d'applications : le comportement de l'application pour le point d'entrée "fonctionnel", les besoins utilisateurs pour le point d'entrée "tâches" et le placement des éléments de l'IHM et l'utilisabilité de celle-ci pour le point d'entrée "interface graphique".

Nous retranscrivons notre cadre d'étude décrit dans la section 1 du chapitre 1 dans le tableau 2.1. Ce-dernier est organisé en trois parties : point(s) d'entrée, modèle(s) utilisé(s) et résultat (de la composition). Pour les points d'entrées, nous étudions une composition multi-niveau guidée par la manipulation des interfaces graphiques. Cependant nous voulons aussi pouvoir changer de point d'entrée de la composition en prévision de nos expérimentions qui nous encouragerons peut-être à utiliser plusieurs points de vue simultanément. Nous comptons aussi étendre nos travaux par la suite à différentes exploitations de nos résultats. C'est pourquoi nous mettons les points d'entrée "fonctionnel" et "tâche" entre parenthèses. Pour les modèles utilisés, nous voulons exploiter tous les points de vue pour la composition elle-même afin de la faciliter et de produire une application opérationnelle. Pour le résultat de la composition, nous visons la réutilisation plutôt que la génération car nous ne supposons pas avoir accès au code source (ni à ce qui permettrait de le générer). Nous présentons donc les travaux connexes à notre étude que nous placerons dans un tableau similaire.

| **Travaux** | **Points d'entrée** | | | **Modèles utilisés** | | | **Résultat** | |
|---|---|---|---|---|---|---|---|---|
| | **Fonctionnel** | **Tâche** | **Graphique** | **Fonctionnel** | **Tâche** | **Graphique** | **Réutilisation** | **Génération** |
| ***Cadre d'étude*** | ***(x)*** | ***(x)*** | ***x*** | ***x*** | ***x*** | ***x*** | ***x*** | |

TABLE 2.1 – Grille d'évaluation des travaux

### 2.2.1 Composition d'application manipulant le Noyau Fonctionnel des applications

En considérant les différents modèles pour la représentation du Noyau Fonctionnel d'une application, nous étudions les compositions relatives à ces descriptions.

#### Compositions limitées à la partie fonctionnelle

Il s'agit des compositions de services à travers les orchestrations de services et les compositions dans les approches à composants à travers les assemblages de composants.

En ce qui concerne les Web Services WS-* ou les Web Services sémantiques OWL-S, leur composition passent par les orchestrations qui permettent de mettre en relation plusieurs services et d'exposer le résultat comme un nouveau service. Pour les deux types de Web Services, les mêmes principes sont

présents pour décrire leur composition avec la possibilité d'exécuter les opérations des services en séquences, en parallèles, en utilisant des conditions, des boucles etc. Pour les Web Services WS-*, leurs orchestrations s'effectuent à l'aide BPEL4WS [49], un standard OASIS dont la spécification finale se nomme WS-BPEL [39]. L'orchestration permet de mettre en relation plusieurs Web Services en précisant d'une part le flux de contrôle et d'autre part le flux de données. Ces orchestrations sont clairement définies de bout en bout avec une et unique activité d'entrée (point de départ de l'orchestration) et une et unique activité de sortie (point de fin de l'orchestration). La description OWL-S permet de décrire directement les relations entre différents services d'une manière très proche à WS-BPEL. Effectivement, à travers le modèle du service (*service model*), il est possible de décrire le service comme un processus qui peut être de trois types : un processus atomique, qui ne possède pas de sous-processus, un processus composite qui va être composé de sous-processus dont nous allons pouvoir décrire la séquence, le parallélisme, les boucles etc., et un processus "simple" qui est un processus abstrait non directement exécutable, permettant de donner une vue simplifié d'un processus atomique ou composite. Certains travaux comme [61] permettent d'exploiter les Web Services OWL-S pour déduire une composition automatiquement à partir d'un problème donné. A partir d'un problème de composition de Web Services et un état initial donné, ces travaux permettent de transformer ce problème en un problème de planification de tâches et de produire automatiquement une composition "préféré" exploitant ainsi les descriptions sémantiques des services.

Les compositions de composants logiciels passent par la construction d'assemblages de composants. Ces assemblages décrivent les composants composites qui, comme pour les approches d'orchestrations de services permettent de connecter (à travers leur interfaces logicielles, une interface logicielle fournie étant connecté à une interface logicielle requise) plusieurs composants entre eux afin d'en former un nouveau. Ce dernier expose les interfaces logicielles requises nécessaires à son opérationnalisation et les interfaces logicielles fournies relatives aux fonctionnalités de plus haut niveau qu'il propose (cf. Fractal [3], SCA [2, 14] et SLCA [8]). Afin d'établir une telle composition, un langage de composition est nécessaire. Ce langage va permettre de mettre en place la "colle" (*glue code*) nécessaire à la communication entre composants. Des composants spéciaux sont aussi utilisés et sont appelés les Connecteurs [54]. Ils peuvent être classés en différentes catégories [54]. Leurs principales fonctions sont de transmettre les données d'un composant à un autre, de prendre en charge la coordinations entre composants, de convertir des données ou des événements etc.

La principale différence entre les assemblages de composants et les orchestrations de services est que dans la plupart des cas les composants cachent le flux de données à l'intérieur d'eux-mêmes. Dans les deux cas, les éventuelles interfaces graphiques associées aux services ou aux composants ne sont pas traitées. A l'inverse d'autres approches répercutent la manipulation des éléments fonctionnels sur les interfaces graphiques.

#### Compositions de la partie fonctionnelle répercutées sur les interfaces graphiques

Le projet ServFace [56, 41], à partir d'une composition de Web Services, vise à obtenir l'application composée correspondante par génération. La méthodologie définie dans ce projet est basée sur trois étapes [45] : l'annotation des services (*service annotation*), la composition de services (*service composition*) et la génération totalement automatisée et dynamique de l'application (*fully automatic runtime generation*). La première étape nécessite l'intervention d'un expert pour annoter les services avec des éléments d'interface graphique utilisant le modèle MARIA [35]. Ensuite, la seconde étape de composition de services va permettre le développement de l'application ainsi composée. A partir des annotations pré-ajoutés aux services, l'utilisateur peut manipuler les interfaces graphiques de chacun

des services qu'il veut utiliser dans la composition à l'aide d'un éditeur visuel le *Visual Composition Editor*, avec pour résultat la production du Modèle de la Composition d'Application (*Composition Application Model*). A partir de ce modèle, la troisième étape génère l'exécutable de l'application à l'aide d'une transformation *model-to-code*.

Dans ALIAS [48, 57], le but est similaire aux travaux précédents, puisqu'il s'agit d'obtenir une application et notamment l'interface graphique de l'application à partir d'une composition fonctionnelle. Les informations nécessaires pour la composition sont décrites dans le méta-modèle ALIAS. Ce méta-modèle s'appuie sur une représentation de l'interface graphique (abstraction des interfaces graphique des applications à composer) et de la partie fonctionnelle des applications vu comme des composants. A partir d'un assemblage de composants, ces travaux composent la partie graphique de l'application résultante en manipulant les liens opérationnels présents entre la partie graphique et les fonctionnalités des applications existantes. Ils travaillent sur deux types de liens, les liens d'événements décrivant le déclenchement d'une opération et les liens de données décrivant les échanges de données entre l'interface utilisateur et les fonctionnalités. Le résultat de la génération est opérationnelle car les liens entre l'interface utilisateur et les fonctionnalités sont préservés durant le processus de composition.

L'utilisation des "interfaces transparentes" [47] permettent d'effectuer le même type de composition, *i.e.*, basée sur la composition de Web Services. Ces travaux utilisent les principes de la Programmation Orientée Aspect (Aspect-Oriented Programming - AOP) [10] en composant une interface logicielle principale par l'ajout de services volatiles. Un service volatile est un service qui est offert sur une courte période de temps. Leur proposition repose sur des vues abstraites des données (*Abstract Data Views - ADVs*) pour décrire les interfaces logicielles des services volatiles. En utilisant un point de coupe dans ces vues abstraites, ils intègrent les fonctionnalités volatiles dans la description de l'interface logicielle principale puis ils utilisent des transformations XSLT pour obtenir l'interface graphique finale de la composition.

Enfin, des travaux mettent en place une approche de composition "à la volée" (à la conception et/ou à l'exécution) basée sur des web services [64]. Utilisant une approche à base de composants, ils définissent un modèle de composant en deux parties : la partie "interfaces" et la partie implémentation. L'implémentation est basée sur un Web Service et met en place le pattern Modèle-Vue-Contrôleur [15]. La partie "interfaces" du composant fournit deux interfaces : une interface de "programmation" ayant accès directement au "contrôleur" de la partie implémentation et une interface "utilisateur", ou interface graphique, directement liée à la "vue" de la partie implémentation. L'interface de "programmation" expose la logique fonctionnelle du composant tandis que l'interface "utilisateur" d'une part, répond aux actions utilisateurs et invoquent les fonctionnalités correspondantes, et d'autre part, permet la configuration de l'interface graphique du composant (taille, couleur. . . ). Pour composer ces composants "à la volée", ils définissent trois types de connecteurs pour lier les interfaces de "programmation" des composants entre eux :

- le connecteur "simple" (*simple connector*) permettant à deux composants d'avoir une interaction directe,
- le connecteur "de données" (*data connector*) permettant de résoudre les potentiels problèmes de non correspondance entre les formats de données et
- le connecteur "flux de contrôle" (*flow connector*) permettant de connecter plus de deux composants.

Pour la composition des interfaces "utilisateurs" des composants, tous les éléments graphiques des

composants impliqués dans la composition sont réunis dans une nouvelle interface graphique. Afin de manipuler ces éléments graphiques, les auteurs ajustent alors la configuration (à travers l'interface "utilisateur" des composants) afin d'en modifier la structure, la présentation...

### 2.2.2 Composition d'application manipulant l'arbre de tâches des applications

Nous avons vu dans les sections précédentes que certaines approches pour construire l'interface graphique d'une application avaient pour point de départ la conception de l'arbre de tâches associé à celle-ci.

Cette conception apporte des informations sur les actions permises par l'application, informations qui peuvent être exploitées afin de composer les applications.

Afin d'effectuer une telle composition, certains travaux mettent en place un travail de composition à partir de la description de cet arbre de tâches.

Dans le projet ServFace [41], nous avons vu qu'une première approche [56] pour construire une application était de partir d'un assemblage de services annotés à l'aide de morceaux d'interfaces abstraites pour ensuite générer l'interface graphique finale de l'application. Toujours en utilisant MARIA [35] comme modèle de description d'interfaces graphiques pour annoter les web services, ce projet propose une deuxième approche [44] qui est basée sur la génération d'une application à partir d'un modèle des tâches. Cette approche comporte deux étapes [58, 59]. Après l'annotation des Web Services avec des morceaux de description d'interfaces en utilisant MARIA, la première étape permet à partir d'un groupe de personnes expertes de constituer un arbre de tâches correspondant à l'application désirée. Chaque tâche est liée à l'un Web Service annoté. La deuxième étape permet le passage du modèle des tâches, *i.e.*, le plus abstrait du *Cameleon Reference Framework (CRF)*, à l'interface finale, *i.e.*, au modèle le plus concret. Ce passage se fait par transformations de modèles afin d'obtenir l'application.

Une approche orientée composants des tâches est effectuée dans [23]. Le principe est de rapprocher le monde des tâches que nous trouvons dans les approches de description d'interface graphique avec le monde de la programmation orientée objet. Ces composants sont construits suivant un processus en cinq étapes. Ces composants de tâches sont constitués d'un arbre de tâches annoté avec les signatures des méthodes qui sont associées à chaque tâche. Puis à partir de cette arbre annoté, les squelettes (et la documentation associée) des méthodes sont générés, puis leur implémentation complétée. Les composants sont alors livrés avec leur documentation et leur modèle de tâches lié au code même des composants.
C'est à partir de tels composants, dans cette approche orientée tâches, qu'une composition de composants dirigée par la fusion d'arbre de tâches est proposé dans [53]. Le principe de composition est le même que celui que l'on peut retrouver dans ComposiXML [52] (cf. section 2.2.3). En considérant que les descriptions de l'arbre de tâches effectuée généralement avec un langage de type XML sont elles-mêmes un arbre, il est possible de leur appliquer des opérateurs utilisés sur les arbres algébriques. Dans [53], il est plus particulièrement décrit l'union entre deux arbres. Après transformations des arbres de tâches en arbres algébriques, et après l'identification de sous-arbres similaires correspondants à des groupes de sous-tâches, l'opérateur d'union est appliqué en créant une nouvelle tâche de plus haut niveau permettant de regrouper les tâches similaires. Ensuite cette nouvelle tâche est placée avant l'activation des autres tâches des deux arbres à fusionner, afin de respecter l'ordre temporel des tâches. Finalement, ces travaux s'appuient sur les composants de tâches afin de générer un nouveau composant correspondant à la fusion des arbres de tâches.

Une autre manière d'approcher la composition à partir de l'arbre de tâches est d'utiliser les principes de planification comme dans l'approche proposée par Compose [46]. A partir d'une requête en langage naturelle effectuée par l'utilisateur, des algorithmes de planification automatique permettent de générer la séquence d'actions à effectuer pour atteindre le but exprimé par la requête, le plan d'actions. Cette séquence d'actions est alors transformée grâce à différentes transformations de modèles entre la machine à états représentants le plan d'actions et les modèles présents dans le *CRF* pour atteindre la génération de l'interface finale de l'application. Le coeur de Compose permet notamment de transformer le plan d'actions en un arbre de tâches. Chaque tâche va alors être associée à un COMET [43] qui est un widget s'adaptant au contexte d'utilisation, défini par une tâche et embarquant en son sein toutes les réifications que nous pouvons trouver dans le *CRF*, de la tâche au modèle d'interface abstraite, puis concrète puis finale. A partir de l'arbre de tâches, un assemblage de COMET est généré, permettant de générer à son tour l'interface finale de l'application dont le but a été défini par l'utilisateur.

### 2.2.3 Composition d'application manipulant l'interface graphique des applications

L'interface graphique d'une application peut être décrite à différents niveaux d'abstraction. Des travaux de composition sont eux aussi effectués à partir de ces différentes niveaux d'abstraction.

Dans UsiXML [31], la composition [52, 51] est basée sur les opérations possibles sur des arbres algébriques, les interfaces graphiques étant représentées par des arbres. En effet UsiXML est un langage XML, les descriptions des interfaces graphiques sont donc des arbres. Ces travaux fournissent un ensemble d'opérateurs (cf. figure 2.3) unaires (prenant en entrée un arbre UsiXML et un pattern d'exploration) ou binaires (prenant en entrée deux arbres UsiXML) afin de pouvoir manipuler les interfaces des applications. Ces manipulations peuvent s'appliquer sur la représentation concrète de l'interface [52] ou sur la représentation abstraite [50] afin de pouvoir travailler sur la multi-modalité des interfaces.

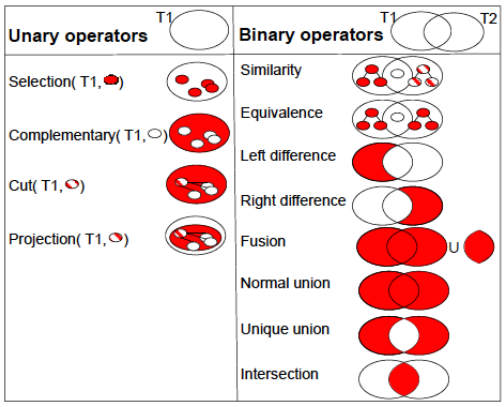



FIGURE 2.3 – Opérateurs unaires et binaires pour effectuer la composition avec UsiXML tiré de [52].

WinCuts [63] et UI façades [62] sont des solutions pour manipuler directement le rendu graphique effectué par le système d'exploitation de l'utilisateur. Ces deux approches proposent de pouvoir reco-

pier le rendu graphique d'une région d'une application afin soit de pouvoir la partager dans le cadre de WinCuts, soit de pouvoir combiner ces régions dans une nouvelle interface graphique. Le système de UI façades permet aussi de pouvoir remplacer certains éléments graphiques par d'autres éléments pré-construits qu'il est possible de personnaliser. Ce système agit comme un calque transparent par dessus le système de fenêtrage du système d'exploitation afin de pouvoir rediriger les événements d'entrées et de dupliquer le contenu des régions sélectionnées afin de les reporter dans d'autres fenêtres "façade".

Les COTS-UI [42] sont exploités dans une approche appliquée aux Systèmes d'Informations basés sur le Web. Ces travaux définissent des composants d'interface graphique appelés *cotsget*. Notons que ces travaux, bien que se basant sur une approche à composant, ne traitent que de la partie interface graphique. Un tel composant est constitué d'une liste de services fournis et une liste de services requis. A partir de cette définition d'un *cotsget*, il est possible de construire une interface graphique. Cette construction se base sur une *Cotsget Architecture* qui est constituée d'un ensemble de composants *cotsget* et d'un ensemble de "dépendances". Celles-ci représentent les connecteurs entre les *cotsget* ayant un rôle "in" comme entrée pour un service fourni par un *cotsget* et un rôle "out" comme sortie provenant d'un service requis par un *cotsget*. A partir de cette architecture, l'approche permet de générer l'interface graphique correspondante en utilisant un service qui permet de trouver les bons services satisfaisant la description de l'architecture.

Le projet CRUISe (*Composition of Rich User Interface Services*) [60] permet de composer des interfaces graphiques web. Ces travaux appliquent les principes des Architectures Orientées Services [5] aux interfaces graphiques, en encapsulant des morceaux d'interfaces graphiques dans un service appelé UIS (*User Interface Service*). Notons que ces travaux, bien que se basant sur une approche orientée service, ne traitent que de la partie interface graphique. Le principal apport de ces services est de faciliter l'interopérabilité à travers une interface UIS générique. Ainsi à partir d'une composition CRUISe construite par un développeur, le "Service d'Intégration" (*Integration Service*) va aller rechercher les UIS correspondant à la description de la composition dans un dépôt UIS. Le "Générateur d'Application" (*Application Generator*) va alors mettre en œuvre la composition demandée par le développeur pour construire une application qui lui sera disponible dans l'environnement d'exécution CRUISe (*CRUISe Runtime*).

### 2.2.4 Compositions présentes sur le Web

Dans le domaine du Web, les travaux autour de la composition d'application se font directement au niveau "interface finale" et des services.

Les travaux les plus répandus concernent les mashups c'est-à-dire des applications regroupant un ensemble de "petites" applications proposant un service particulier (la météo, les news, pages jaunes, programmes télé, etc). Il existe deux types de travaux qui se différencient par le fait d'utiliser ou non les flots de données au niveau de la composition et qui répondent en réalité à deux besoins différents :

1. la composition de données, *i.e.*, partager des données et faire différents traitements sur ces données et
2. la juxtaposition de données, *i.e.*, avoir un accès rapide à différents services sur une seule page.

#### Composition de données

Certains applications Web comme IFTTT.com [2] permettent de mettre en place des chorégraphies entre services sans fournir pour autant d'interfaces graphiques pour les manipuler. Celui-ci est basé sur la phase "If This Then That" où un service va service de déclenchement (le "This") et un second service va servir d'action (le "That"). D'autres applications Web vont permettre de mettre en place le même genre de chorégraphie de services mais de manière plus graphique avec les sorties de services reliés directement par "des flèches" aux entrées d'autres services. Ces applications comme Popfly [3] de Microsoft ou Yahoo Pipes [4] permettent la mise en place d'une interface graphique (l'interface du dernier service utilisé) pour manipuler la chorégraphie ainsi construite.

Ces compositions sont essentiellement des compositions manipulant le noyau fonctionnel.

#### Juxtaposition de données

IGoogle [5] ou Netvibes [6] utilisent les mashups pour répondre au besoin d'accès rapide aux services. Pour ce type de mashups, la seule composition possible est en réalité le regroupement de plusieurs applications dans des catégories définies par l'utilisateur et mettre côte à côte les interfaces graphiques de ces applications. Ces portails Web ne permettent pas d'effectuer d'échanges de données entre les différents services utilisés.

Ces compositions sont essentiellement des compositions manipulant l'interface graphique.

#### Composition et Juxtaposition de données

Enfin, certaines applications Web (voire des clients lourds) mettent en place les techniques de *Web Scrapping*. Cette technique consiste à récupérer des zones de sites Web et à afficher toutes ces zones sur une seule et même page. La différence avec les mashups est que nous n'avons pas un regroupement de "petites" applications sur une même page mais bien des morceaux de sites web se mettant à jour comme sur les sites web originaux d'où elles ont été extraites.

## 2.3 Synthèse de l'état de l'art

Le tableau 2.2 résume cette étude bibliographique. En dernière ligne, nous avons reporté ce que nous cherchons, comme exprimé en début de chapitre. Les compositions reposent sur des descriptions des applications. Ces descriptions dissocient deux parties de l'application, sa partie fonctionnelle et son interface graphique. La construction d'une interface graphique passent par différentes étapes dont la principale est la constitution de l'arbre de tâches associé à l'application. C'est ainsi que plusieurs points de vue sont disponibles pour décrire une application : la description de son Noyau Fonctionnel, la description de son interface graphique et la description de son arbre de tâches. C'est autour de ces points de vue que les compositions sont effectuées.

Certains travaux utilisent même une combinaison de ces points de vue afin de réaliser la composition d'applications. Ces travaux montrent qu'il est nécessaire de lier les différents modèles des points

2. https ://ifttt.com/
3. http ://www.popfly.ms/microsoft-popfly/
4. http ://pipes.yahoo.com/
5. http ://www.google.com/ig
6. http ://www.netvibes.com/

de vue d'une application afin de pouvoir exploiter un maximum d'informations pour mener à bien la composition. Par rapport à notre cadre d'étude (c.f. la section 1 du chapitre 1), nous identifions quatre travaux assez proches. Ils sont identifiables sur la table 2.2 car ils totalisent au moins quatre "x" pour les points d'entrée et les modèles utilisées et ils réutilisent (au moins partiellement) les applications initiales. Ces travaux sont :

- ServFace [56, 41, 35, 44, 58, 59] dont la différence par rapport à notre cadre d'étude est que ServFace génère les interfaces graphiques,
- composition "à la volée" basée sur des web services [64] dont la différence par rapport à notre cadre d'étude est leur non prise en compte des tâches,
- Compose [46] dont la différence par rapport à notre cadre d'étude est son (unique) point d'entrée, les tâches et
- CRUISe [60] dont la différence par rapport à notre cadre d'étude est leur non prise en compte des tâches.

Les travaux proposant une composition basée sur les interfaces graphiques des applications proposent soit la manipulation directe du système de fenêtrage du système d'exploitation de l'utilisateur, soit une composition basée sur la description des interfaces abstraites ou concrètes à travers des opérations sur les arbres algébriques, arbre représentant la hiérarchie de l'interface graphique des applications.

Notre originalité est donc de **considérer les liens existants entre la partie graphique, les tâches et la partie fonctionnelle des applications à composer**. Notre proposition est une composition plus fine, à travers une manipulation directe des éléments des interfaces graphiques et **une aide apportée au meneur de la composition lors de la sélection** de ces éléments graphiques. Ces algorithmes de sélection sont basés sur l'exploitation des informations à disposition à partir des autres points de vue sur l'application. Notre but est d'atteindre une composition consistante et une application résultante opérationnelle. C'est pour cette raison que nous proposons **une composition basée sur des substitutions** entre éléments graphiques des applications à composer.

| Travaux | Points d'entrée | | | Modèles utilisés | | | Résultat | |
|---|---|---|---|---|---|---|---|---|
| | **Fonctionnel** | **Tâche** | **Graphique** | **Fonctionnel** | **Tâche** | **Graphique** | **Réutilisation** | **Génération** |
| Approches à Services : BPEL4WS [49] - BPEL [39] - Composition de Web Service OWL-S [61] | x | | | x | | | x | |
| Approches à Composants : Fractal [3], SCA [2, 14] et SLCA [8] | x | | | x | | | x | |
| ServFace [56, 41, 35, 44, 58, 59] | x | x | | x | x | x | x (services) | x (interfaces graphiques) |
| ALIAS [48, 57] | x | | | x | | x | x (services) | x (interfaces graphiques) |
| "interfaces transparentes" [47] | x | | | x | | x | x (services) | x (interfaces graphiques) |
| composition "à la volée" basée sur des web services [64] | x | | x | x | | x | x | |
| composition de tâches [23] | | x | | x | x | | | x (code à compléter) |
| fusion d'arbres de tâches [53] | | x | | x | x | | x | |
| Compose [46] | | x | | x | x | x | x | |
| ComposiXML [52] | | | x | | | x | x (AUI ou CUI) | |
| WinCuts [63] | | | x | | | x | x | |
| UI façades [62] | | | x | | | x | x | |
| COTS-UI [42] | | | x | x | | x | | x |
| CRUISe [60] | x | | x | x | | x | x | |
| ***Cadre d'étude*** | ***(x)*** | ***(x)*** | ***x*** | ***x*** | ***x*** | ***x*** | ***x*** | |

TABLE 2.2 – Caractérisations des approches de composition

# Deuxième partie

# Contribution

CHAPITRE 3

# Modèle d'application pour une composition multi-modèles

## Sommaire



NOUS présentons dans ce chapitre notre modèle de description des applications. Cette description fait référence aux entités informatiques constituant l'application modélisée. Nous nous sommes fixés comme objectif d'avoir un modèle :

- qui prenne en compte les différents points de vue utilisées pour la composition d'applications (cf. chapitre 2),
- qui permette la manipulation de parties des applications afin de pouvoir sélectionner/extraire des morceaux provenant de différentes applications pour les relier en une nouvelle application,
- qui permette la manipulation de parties des applications afin de limiter l'écriture de code spécifique et
- qui capitalise sur les travaux antérieurs pour déterminer les modèles à réutiliser tout en précisant les liens établis entre eux.

Notre contribution se situe donc dans la mise en relation entre différents modèles provenant de points de vue différents : l'aspect fonctionnel, l'aspect interaction graphique et l'aspect besoin des utilisateurs à travers les tâches à accomplir. Nous présentons donc dans ce chapitre les modélisations de ces trois points de vue, puis nous décrivons leurs relations constitutives de notre modèle pour la composition d'applications.

## 3.1 Modèle opérationnel d'une Application

Toute application est constituée d'entités informatiques qu'il faudra manipuler lors de la composition. Il s'agit donc de pouvoir sortir de leur contexte initial une partie de ces entités pour les connecter ensemble. Ainsi pour chaque entité, il faut déterminer ce dont elles ont besoin pour fonctionner et ce qu'elles peuvent produire comme résultats. Nous n'avons pas d'autres hypothèses sur ces entités. En particulier, nous ne supposons pas avoir accès au code source des applications. Ces entités correspondent à une représentation par composants (cf. chapitre 2).

Ces entités d'une application, que nous pourrions appeler par métaphore "composants", représentent ainsi les morceaux de l'application que nous manipulons. Nous appellerons ces entités par la suite "élément logiciel" (et non pas composant pour ne pas faire d'amalgame entre le modèle et les implémentations possibles). Nous basons notre démarche de composition sur ces éléments logiciels. De fait, notre modèle opérationnel s'inspire fortement des modèles à composant non hiérarchique. Nous n'explicitons ici que les informations nécessaires pour la composition. Ces informations sont parfois non présentes dans les modèles à composant. Autre conséquence, les modèles à composants ont souvent une description de l'assemblage qui constituent une application. Cette description pourrait servir de base (parfois à compléter) à l'instanciation de notre modèle opérationnel pour cette application.

La granularité de cette composition dépend donc de la manière dont l'application a été conçue. Si elle est basée sur un modèle de conception tel que MVC [15] avec une claire séparation des préoccupations alors la granularité de manipulation de l'application sera assez fine. Il sera possible de manipuler et (re)connecter de petites parties de l'application. Si au contraire, l'application n'apparaît pas être conçu avec des éléments logiciels spécialisés (un élément logiciel correspondant à plusieurs fonctionnalités), il y a alors une plus grande chance que la granularité de manipulation de l'application soit assez grossière et que donc ces éléments logiciels réalisent une grande sous-partie de ses fonctionnalités avec peu d'éléments.

Dans ce modèle fonctionnel de l'application, tous les éléments logiciels de l'application y sont représentés, ainsi que les liens opérationnels entre eux. Ces éléments logiciels peuvent appartenir à la partie fonctionnelle de l'application mais aussi à l'interface graphique de l'application voire aux deux. Nous pouvons voir une première partie de cette représentation opérationnelle sur la figure 3.1 qui illustre les liens opérationnels des éléments logiciels de l'application "Maps" de l'étude de cas (cf. section 1.4).

Soit *APPLI* une application. Une application est alors constituée d'éléments logiciels (les composants de l'application) : $APPLI = \{appelem_i\}$, l'ensemble de tous les éléments logiciels constituants une application.

Tout élément logiciel d'une application aura un identifiant unique. Nous définissons cette propriété de la manière suivante [1] :

> $appElemId : APPLI \rightarrow String$ alors,
> – si *aeid* est l'identifiant d'un élément $appelem$ de l'application, $appElemId(appelem) = aeid$
> – $\forall appelem_1, appelem_2 \in APPLI, appElemId(appelem_1) \neq appElemId(appelem_2)$

1. Par la suite, nous utiliserons l'ensemble $String$ pour représenter l'ensemble des chaînes de caractères non vide

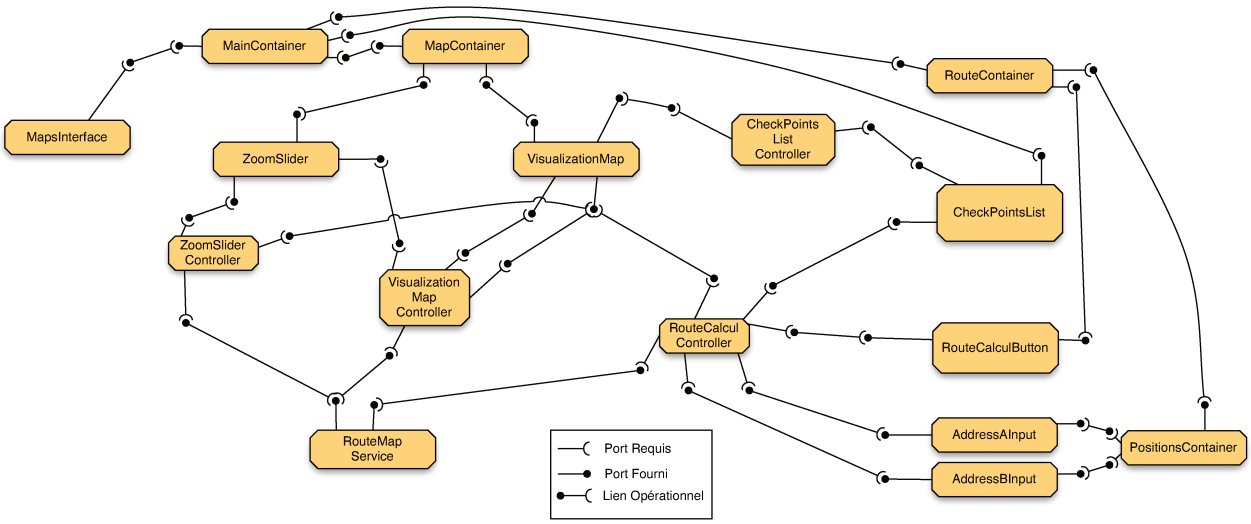



FIGURE 3.1 – Illustration des éléments logiciels et des liens opérationnels entre ces éléments de l'application "Maps".

### 3.1.1 Ports

A la manière des composants, chaque élément logiciel d'application est constitué d'interfaces logicielle que nous appellerons ports. Soit $PORTS$ l'ensemble des ports de tous les éléments logiciels d'applications :

Soit $associateAppElemPort$ la fonction permettant d'associer un port à un élément logiciel. Nous noterons ici que cette association est unique, c'est-à-dire qu'un port ne peut pas appartenir à plusieurs éléments à la fois.

$$associateAppElemPort : APPLI \times PORTS \rightarrow BOOLEAN,$$
$$\forall appelem \in APPLI, \forall port \in PORTS,$$
$$associateAppElemPort(appelem, port) \Leftrightarrow port \text{ est un port de l'élément logiciel } appelem$$

Nous définissons alors une fonction qui permet d'identifier l'élément logiciel de l'application auquel est associé le port passé en paramètre. Nous obtenons :

$$portAssociatedWith : PORTS \rightarrow APPLI,$$
$$\forall port \in PORTS, \exists appelem \in APPLI/$$
$$portAssociatedWith(port) = appelem \Leftrightarrow associateAppElemPort(appelem, port)$$

Afin d'obtenir tous les ports associés à un élément logiciel, nous définissons la fonction suivante.

**Remarque :** Dans le reste du manuscrit, quelque soit l'ensemble $A$, $\mathcal{P}(A)$ représente l'ensemble des partitions possibles de $A$ (y compris la partition vide).

Soit $portsFromAppElem$, la fonction permettant d'obtenir tous les ports associés à un élément logiciel :
$portsFromAppElem : APPLI \rightarrow \mathcal{P}(PORTS)$,
$\forall appelem \in APPLI, portsFromAppElem(appelem) = \{port_j \in PORTS, associateAppElemPort(appelem, port_j) = TRUE\}$

### 3.1.2 Ports Requis et Ports Fournis

C'est à travers les ports que les éléments logiciels d'une application vont être connectés. Ils peuvent donc être un indicateur des échanges d'informations entre les éléments. Certains ports peuvent être utilisés pour obtenir de l'information ou un "service". D'autres peuvent au contraire fournir ces informations. En reprenant la terminologie "composant", chaque port est donc soit requis soit fourni.

En considérant :
$isRequired : PORTS \rightarrow BOOLEAN$
$isProvided : PORTS \rightarrow BOOLEAN$
2 fonctions permettant respectivement de savoir si un port de l'élément logiciel est requis ou fourni par celui-ci, alors :

$\forall p \in PORTS, isRequired(p) \Leftrightarrow \neg isProvided(p)$ (et réciproquement, $isProvided(p) \Leftrightarrow \neg isRequired(p)$)

### 3.1.3 Types de ports

Nous avons besoin d'informations concernant le comportement des différents éléments logiciels de l'application et cela s'effectue à l'aide des renseignements sur les ports de ceux-ci. Chaque port d'un élément logiciel va donc être associé à un TYPE (unique pour le port) qui va nous permettre de connaître le comportement requis ou fourni par ce port.
Soit $PORTSTYPES = \{TRIGGER, INPUT, OUTPUT\}$, l'ensemble des types possibles pour un port.
"*Trigger*" décrit le fait qu'à travers ce port, l'élément peut "appeler" un autre élément. Cela peut être par exemple un bouton qui va déclencher une action particulière ou un élément "observable" qui va notifier tous ses observateurs. Le port de type "*Trigger*" va déclencher des actions d'autres éléments de l'application.
"*Input*" est utilisé lorsque qu'un port permet de récupérer des informations. Les éléments avec un tel port "fourni" peuvent donc être interrogés pour obtenir des informations par les autres éléments de l'application. On peut prendre pour exemple un élément logiciel embarquant un élément graphique "Input Text" qui permet à l'utilisateur de donner de l'information textuelle à l'application. Cet élément logiciel a un port de type "*Input*" (pouvant correspondre à la méthode "$getText()$" ) permettant à d'autres composants de récupérer cette information textuelle afin de la traiter.
Enfin, "*Output*" permet de décrire un port laissant la possibilité de définir une donnée ou des informations. Cela peut correspondre à un port d'un élément logiciel manipulant un élément graphique de sortie pouvant afficher des données à l'écran ou un port d'un élément fonctionnel permettant d'enregistrer les données dans une base de données. C'est à travers ce port que les informations vont pouvoir être passées à l'élément pour qu'il puisse remplir sa fonction.

En considérant :
$typePort : PORTS \rightarrow PORTSTYPES$ la fonction permettant de récupérer le type du port, nous obtenons 3 fonctions permettant de valider le type d'un port :

- Une fonction qui permet de savoir si le port est de type TRIGGER
  $isTrigger : PORTS \rightarrow BOOLEAN$ tel que
  $\forall p \in PORTS, isTrigger(p) \Leftrightarrow typePort(p) = TRIGGER$
- Une fonction qui permet de savoir si le port est de type INPUT
  $isInput : PORTS \rightarrow BOOLEAN$ tel que
  $\forall p \in PORTS, isInput(p) \Leftrightarrow typePort(p) = INPUT$
- Une fonction qui permet de savoir si le port est de type OUTPUT
  $isOutput : PORTS \rightarrow BOOLEAN$ tel que
  $\forall p \in PORTS, isOutput(p) \Leftrightarrow typePort(p) = OUTPUT$

Nous définissons alors une propriété sur les ports mettant en évidence que le port ne peut avoir qu'un seul type :

$\forall p \in PORTS,$
- $isTrigger(p) = \neg isInput(p) \wedge \neg isOutput(p)$
- $isInput(p) = \neg isOutput(p) \wedge \neg isTrigger(p)$
- $isOutput(p) = \neg isTrigger(p) \wedge \neg isInput(p)$

Nous illustrons l'ajout de ce type sur les ports des éléments de l'application "Maps" sur la figure 3.2.

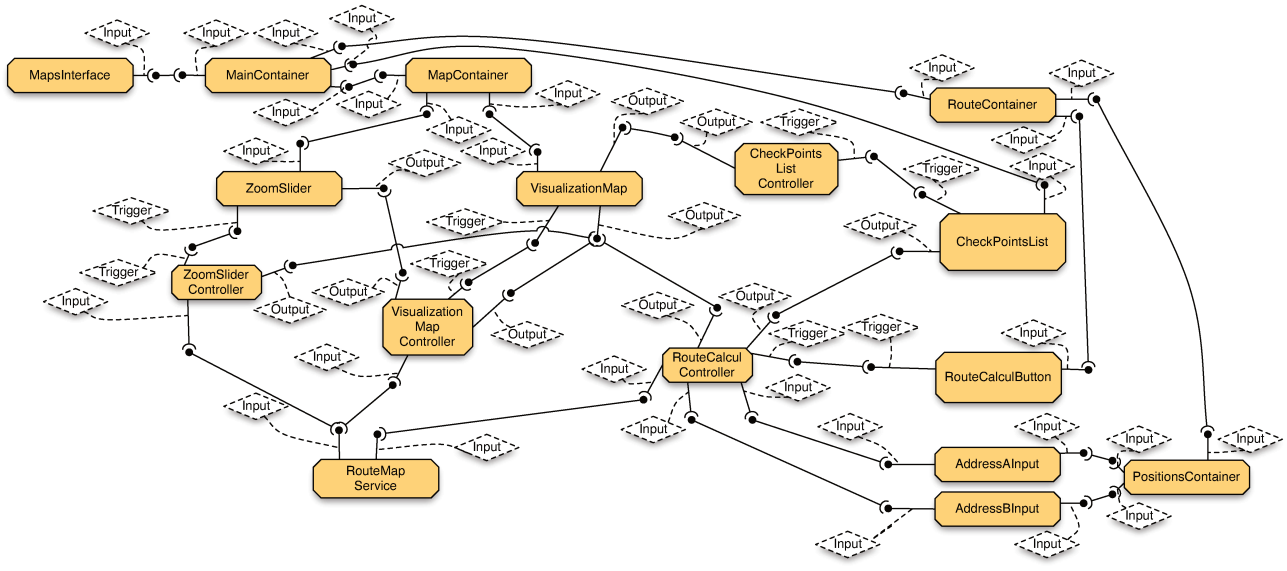



FIGURE 3.2 – Illustration de l'ajout des types sur les ports des éléments logiciels de l'application "Maps".

Par rapport aux modèles à composant, les informations relatives aux types de port ne sont à priori

pas présentes dans les descriptions des assemblages. Ces informations sont donc à préciser explicitement pour chaque application.

### 3.1.4 Rôle d'un port

Chaque port joue un rôle particulier dans l'application. Notre contexte de travail étant la composition dirigée par les interfaces graphiques, nous nous sommes concentrés sur les rôles liées aux interactions graphiques. Nous en avons identifiés deux :

1. le rôle "UI" d'un port s'il intervient **directement dans l'interaction graphique** avec l'utilisateur et
2. le rôle "UI Component" d'un port **s'il retourne** un élément appartenant à l'interface graphique.

Par rapport aux modèles à composant, les informations relatives aux rôles ne sont à priori pas présentes dans les descriptions des assemblages. Ces informations sont donc à préciser explicitement pour chaque application.

#### Rôle "UI" d'un port

Nous faisons cette distinction au niveau des ports et non pas directement au niveau de l'élément de l'application. Effectivement, si nous prenons les éléments logiciels qui permettent de mettre en place l'interface graphique de l'application, certains peuvent exposer des ports qui ne sont pas en rapport direct avec l'interaction utilisateur (comme une fonctionnalité d'un contrôleur qui serait présent dans l'élément logiciel et exposé par celui-ci). Nous utilisons alors ce rôle pour bien distinguer à propos d'un seul élément d'application, les ports qui lui permettent d'avoir une interaction avec l'utilisateur de ses autres ports ayant un autre rôle ....

> Nous définissons la propriété d'un port permettant de savoir s'il a un rôle UI ou non :
>
> $isUIPort : PORTS \rightarrow BOOLEAN$ tel que
> $\forall p \in PORTS, isUIPort(p) \Leftrightarrow$ le port $p$ a le rôle "UI", d'interaction avec l'utilisateur à travers l'interface graphique de l'application.

#### Ports "UI Component"

Dans la manipulation des éléments logiciels de l'application, nous avons besoin d'identifier les éléments ayant un port fourni permettant de récupérer (c'est-à-dire de type *Input*) un élément graphique (défini dans la section suivante). Pour cela, nous définissons l'annotation "UI Component" sur ces ports pour les distinguer des ports "UI" permettant l'interaction direct avec l'interface. Ces ports particuliers permettront dans la phase de sélection de distinguer les éléments graphiques "sélectionnables" des éléments graphiques "non sélectionnables" (cf. figure 3.3)

> Nous définissons une propriété d'un port permettant de savoir s'il a un rôle "UI Component" ou non :
>
> $isUIComponentPort : PORTS \rightarrow BOOLEAN$ tel que
> $\forall p \in PORTS, isUIComponentPort(p) \Leftrightarrow$ le port $p$ a le rôle "UI Component", permettant de récupérer un élément graphique de l'interface graphique de l'application.

Notons l'implication suivante :
$\forall p \in PORTS, isUIComponentPort(p) \Rightarrow isProvided(p) \wedge isInput(p)$

**Illustration des rôles "UI" et "UI Component"**

La figure 3.3 montre un exemple de l'ajout de cette information sous forme d'annotations. Nous constatons qu'elle permet clairement de faire la différence entre les éléments logiciels manipulant les éléments graphiques de l'application ("UI Component"), de ceux conduisant à une modification "immédiate" de l'interface graphique ("UI"), et les autres éléments de l'application (manipulant notamment la partie fonctionnelle de l'application).

Par exemple, le port de "VizualisationMap" connecté à "MapContainer" est annoté "UI Component" car il fournit à "MapContainer" l'entité graphique (le *panel* portant la carte) à ajouter dans la *fenêtre* de l'application. Quant au port de "AddressAInput" connecté à "RouteCalcul Controller", il est annoté "UI" car il fournit le texte tapé par l'utilisateur dans le champ prévu pour saisir l'adresse de départ.

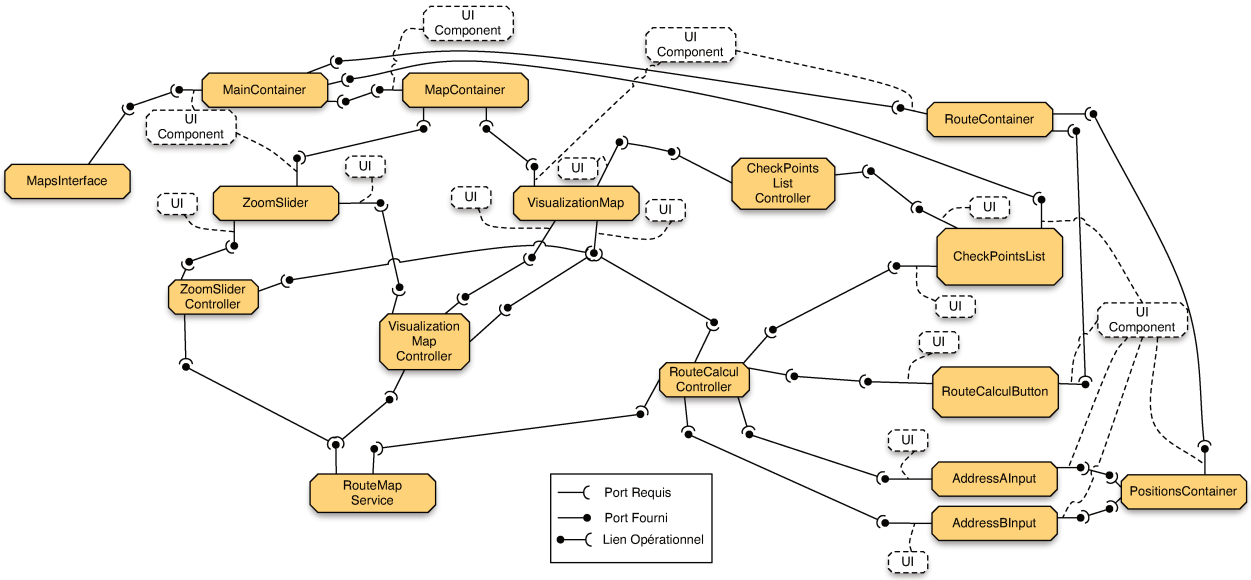



FIGURE 3.3 – Illustration de l'ajout des rôles "UI" et "UI Component" sur les éléments logiciels de l'application "Maps".

### 3.1.5 Connexions de ports

Afin de constituer une application, il faut nécessairement une connexion entre ceux-ci. Cette connexion à l'instar de l'approche à composants, s'effectue à travers la connexion des ports des éléments logiciels de l'application (cf. figure 3.1). Une connexion entre deux ports peut s'effectuer si les deux ports sont de même type et si l'un est requis l'autre doit être fourni (et vice-versa).

Nous définissons la propriété de connexions entre deux ports de deux éléments de l'application :
$areConnected : PORTS \times PORTS \rightarrow BOOLEAN$, la fonction permettant de savoir si 2 ports sont

connectés :
$\forall p,q \in PORTS, areConnected(p,q) \Leftrightarrow p$ et $q$ sont connectés : $p$ est connecté à $q$ et $q$ est connecté à $p$

$areConnected(p,q) \Rightarrow ((isRequired(p) \wedge isProvided(q)) \vee (isProvided(p) \wedge isRequired(q))) \wedge typePort(p) = typePort(q)$

Ainsi, à partir d'un port donné, nous définissons la fonction *connexions* qui identifie tous les ports qui y sont connectés :
$connexions : PORTS \rightarrow \mathcal{P}(PORTS)$ la fonction permettant de connaître les ports connectés à un port donné
$\forall p \in PORTS, connexions(p) = \{q \in PORTS / areConnected(p,q)\}$

## 3.2 Modélisation de l'interface graphique d'une application et de sa "mise en page"

Ayant défini le point de vue opérationnel d'une application, nous présentons maintenant l'aspect interaction graphique.

Soit $APPUI$, l'interface graphique d'une application. $APPUI = \{UIELEM_i\}$ ensemble de tous les éléments constituants l'interface graphique d'une application. Dans l'application, ces éléments graphiques ne prennent pas forcément la forme d'un seul élément logiciel. Dans une application basée composants, il n'est pas rare de retrouver des composants qui regroupent plusieurs éléments graphiques. Ayant besoin de manipuler ces éléments de l'interface graphique de l'application, nous décrivons cette interface en supplément de la description des éléments logiciels de l'application. Nous affirmons donc que : $APPUI \neq APPLI$, c'est-à-dire que nous considérons qu'un élément logiciel peut encapsuler plusieurs éléments de l'interface graphique. A l'inverse, un élément graphique ne peut être encapsulé que par un seul élément logiciel.

Nous prenons ici en compte les éléments de l'interface graphique, c'est-à-dire les éléments de l'application ayant une interaction avec l'utilisateur. Ce qui nous intéresse ici est de modéliser la "mise en page" (*layout*) de l'interface graphique de l'application. Cette modélisation de la "mise en page" a été construite à partir de l'étude de certains langages de programmation et la manière de construire des interfaces graphiques avec ces langages comme Java (Swing) [72], MXML pour Flex [70], XAML pour WPF [68] ... Notre modélisation est à priori extractible par introspection et analyse des interfaces graphiques.

L'interface graphique d'une application est communément décrite de manière hiérarchique décrivant ainsi l'imbrication de certains éléments graphiques dans des "conteneurs". Une fois cette structure en place, nous différencions trois manières de positionner les éléments nous permettant d'exprimer n'importe quel positionnement présent dans les outils de développement :

1. Le "layout absolu" est la mise en page traditionnelle avec l'utilisation de coordonnées absolues. Elle permet de décrire la position d'un composant graphique sur l'axe des abscisses et l'axe des ordonnées par rapport à son conteneur parent. C'est la mise en page la plus flexible car les valeurs des coordonnées ne sont pas fixées et peuvent être aussi bien positives que négatives.
2. Le "layout tableau" consiste à placer les différents composants dans une grille associée à son conteneur parent. Dans cette mise en page, le composant peut alors prendre neuf positions comme décrit dans la figure 3.4.

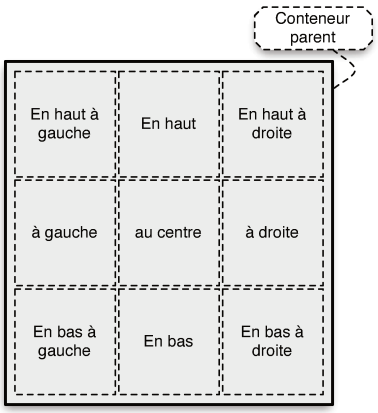



FIGURE 3.4 – Grille de placement d'un élément dans son conteneur parent.

3. Le "Layout Relatif" (cf. figure 3.5) utilise des contraintes pour exprimer le placement entre deux éléments graphiques l'un relativement par rapport à l'autre (l'élément *A* est situé à gauche de l'élément *B*, l'élément *C* est situé au dessus de l'élément *B*). Contrairement aux deux précédentes mises en page, celle-ci ne nécessite pas de définir de conteneur pour positionner les éléments.

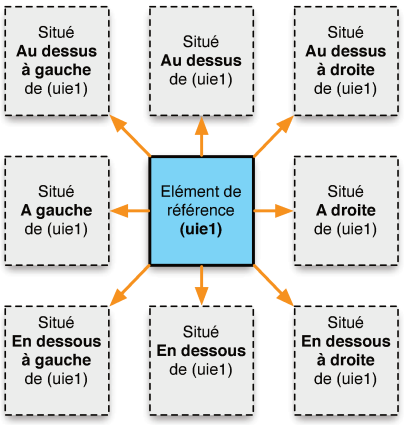



FIGURE 3.5 – Positions relatives possibles entre deux éléments de l'interface graphique.

C'est cette troisième mise en page que nous avons gardé pour modéliser le positionnement de l'élément dans l'interface car c'est la mise en page qui nous paraît la plus pertinente pour l'utilisateur. Enfin, à partir des deux autres mises en page, il existe une transformation pour exprimer les positions des éléments en un positionnement relatif.

### 3.2.1 Structure hiérarchique de l'interface graphique

Pour décrire l'interface graphique de l'application, nous utilisons une structure hiérarchique. A partir d'un élément de l'interface graphique, nous définissons des fonctions pour retrouver son élément

parent (son conteneur) ou ses éléments enfants (ceux qu'il contient). Tous les éléments de l'interface graphique ont un élément parent sauf l'élément "racine" qui est le conteneur principal de l'interface de l'application.

Nous définissons alors les fonctions suivantes :

- $uiElemParent : APPUI \rightarrow APPUI \bigcup \emptyset$
  $\begin{cases} \forall uie1 \in APPUI, \\ uiElemParent(uie1) = uie2 \text{ si } uie2 \text{ est l'élément graphique parent de l'élément graphique } uie1 \\ uiElemParent(uie1) = \varnothing \text{ si } uie1 \text{ n'a pas d'élément graphique parent} \end{cases}$
- $uiElemChildren : APPUI \rightarrow \mathcal{P}(APPUI)$
  $\forall uie_i \in APPUI,$
  $uiElemChildren(uie_i) = \{uie_j \in APPUI / \forall j, uie_i \neq uie_j \wedge uiElemParent(uie_j) = uie_i\}$

Nous définissons également une fonction pour savoir si un élément graphique est l'ancêtre d'un autre :

$isAncestorOf(uie_{parent}, uie_{child}) : APPUI \times APPUI \rightarrow BOOLEAN$
$\forall uie_{parent}, uie_{child} \in APPUI,$
$isAncestorOf(uie_{parent}, uie_{child}) \Leftrightarrow (uie_{parent} = uie_{child}) \vee (uie_{parent} = uiElemParent(uie_{child})) \vee$
$(\exists uie_{path} \in APPUI / uie_{path} \in uiElemChildren(uie_{parent}) \wedge isAncestorOf(uie_{path}, uie_{child}))$

Les deux fonctions *uiElemChildren* et *uiElemParent* nous permettent donc d'obtenir une description hiérarchique de l'interface graphique de l'application illustrée sur la figure 3.6 pour l'application "Maps". Par exemple, l'élément graphique "RouteContainer" correspond à la partie centrale de l'interface graphique illustrée par la figure 1.2. Il contient les deux descriptions ("Label 'Position A'" et "Label 'Position B'") et les deux champs de texte ("AddressAInput" et "AddressBInput"). Ce qui ne se voit pas directement sur l'interface graphique, c'est que ces quatre éléments sont contenus dans "PositionsContainer" qui est lui directement contenu dans "RouteContainer". Ce dernier contient également le bouton pour lancer le calcul ("RouteCalculButton"). "RouteContainer" est donc un ancêtre de "AddressAInput" (*i.e.*, $isAncestorOf(\text{"}RouteContainer\text{"}, \text{"}AddressAInput\text{"}) = true$), mais "RouteCalculButton" n'est pas un ancêtre de "AddressAInput" (*i.e.*, $isAncestorOf(\text{"}RouteCalculButton\text{"}, \text{"}AddressAInput\text{"}) = false$).

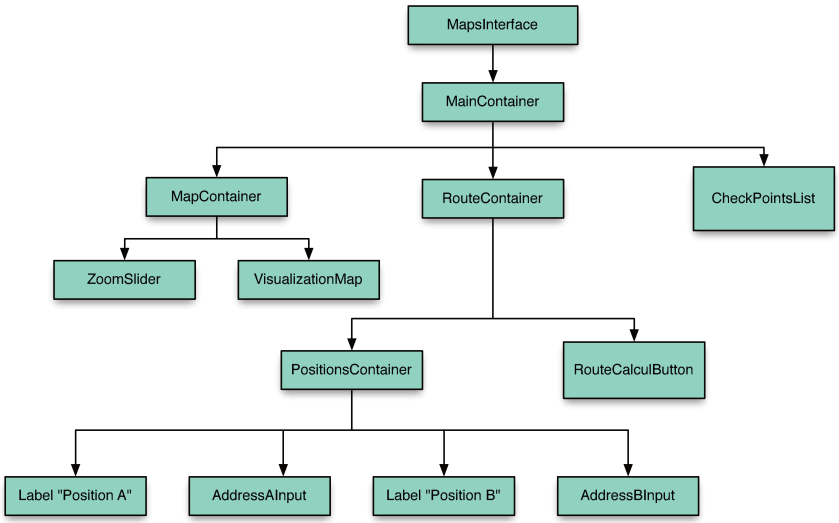



FIGURE 3.6 – Représentation de la structure hiérarchique de l'interface graphique de l'application "Maps".

### 3.2.2 Positionnement des éléments de l'interface graphique

Pour le positionnement des éléments de l'interface graphique de l'application, nous avons opté pour un placement relatif entre deux éléments. Les différentes positions possibles sont donc au nombre de huit comme le présente la figure 3.5.

Les différentes relations de position possibles sont :
$POS = \{isAboveLeftOf, isAboveOf, isAboveRightOf, isOnTheLeftOf, isOnTheRightOf, isBelowLeftOf, isBelowOf, isBelowRightOf\}$

Considérons :
$getPosBetween : APPUI \times APPUI \rightarrow POS \cup \{\varnothing\}$
$\forall uie1, uie2 \in APPUI,$

- $getPosBetween(uie1, uie2) = \varnothing \Leftrightarrow$ aucune relation de position n'existe entre les deux éléments
- $getPosBetween(uie1, uie2) = pos, pos \in POS \Leftrightarrow uie1 \neq uie2 \wedge \exists pos \in POS / pos(uie1, uie2)$, c'est-à-dire qu'il existe une relation de position entre uie2 et uie1

Nous formalisons les différentes positions de la manière suivante :
- $isAboveLeftOf : APPUI \times APPUI \rightarrow BOOLEAN$
  $\forall uie1, uie2 \in APPUI,$
  $isAboveLeftOf(uie1, uie2) \Leftrightarrow uie2$ est situé au dessus à gauche de $uie1$
- $isAboveOf : APPUI \times APPUI \rightarrow BOOLEAN$
  $\forall uie1, uie2 \in APPUI,$
  $isAboveOf(uie1, uie2) \Leftrightarrow uie2$ est situé au dessus de $uie1$
- $isAboveRightOf : APPUI \times APPUI \rightarrow BOOLEAN$
  $\forall uie1, uie2 \in APPUI,$

$isAboveRightOf(uie1, uie2) \Leftrightarrow uie2$ est situé au dessus à droite de $uie1$

- $isOnTheLeftOf : APPUI \times APPUI \rightarrow BOOLEAN$
  $\forall uie1, uie2 \in APPUI$,
  $isOnTheLeftOf(uie1, uie2) \Leftrightarrow uie2$ est situé à gauche de $uie1$
- $isOnTheRightOf : APPUI \times APPUI \rightarrow BOOLEAN$
  $\forall uie1, uie2 \in APPUI$,
  $isOnTheRightOf(uie1, uie2) \Leftrightarrow uie2$ est situé à droite de $uie1$
- $isBelowLeftOf : APPUI \times APPUI \rightarrow BOOLEAN$
  $\forall uie1, uie2 \in APPUI$,
  $isBelowLeftOf(uie1, uie2) \Leftrightarrow uie2$ est situé en dessous à gauche de $uie1$
- $isBelowOf : APPUI \times APPUI \rightarrow BOOLEAN$
  $\forall uie1, uie2 \in APPUI$,
  $isBelowOf(uie1, uie2) \Leftrightarrow uie2$ est situé en dessous de $uie1$
- $isBelowRightOf : APPUI \times APPUI \rightarrow BOOLEAN$
  $\forall uie1, uie2 \in APPUI$,
  $isBelowRightOf(uie1, uie2) \Leftrightarrow uie2$ est situé en dessous à droite de $uie1$

On peut voir un exemple d'application de ces fonctions de positions sur les éléments de l'interface graphique de l'application "Maps" sur la figure 3.7. Comme le montre la figure 1.2, l'élement "Label 'Position A'" est bien à gauche du champ de texte "AddressInputA".

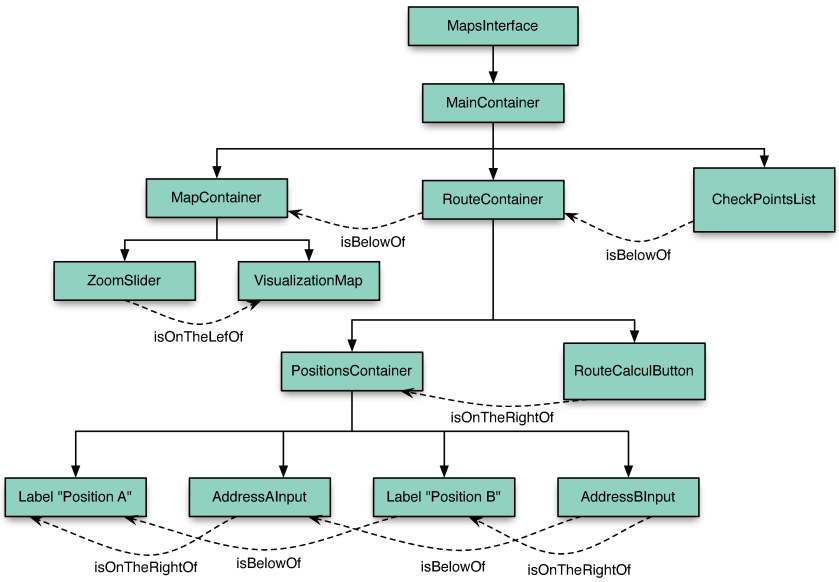



FIGURE 3.7 – Représentation des positions des éléments de l'interface graphique de l'application "Maps".

## 3.3 Arbre de tâches associé à une application

Ayant décrit dans les deux sections précédentes la modélisation fonctionnelle et graphique, nous présentons maintenant la modélisation des besoins de l'utilisateur exprimés sous forme d'un arbre de tâches.

Ce type d'arbre décrit les activités que doit réaliser un utilisateur pour atteindre son but et l'enchaînement de ces activités les unes par rapport aux autres. Les arbres de tâches sont le plus souvent descriptifs et utilisés lors de la conception des applications. Ils sont parfois présents lors de l'exécution des applications. Dans tous les cas, il faudra transformer, manuellement voire automatiquement, ces arbres de tâches dans notre représentation.

Les tâches vont alors représenter les différentes étapes qui, en les enchaînant, permettent d'atteindre un but de plus haut niveau. En nous basant sur l'étude effectuée dans le chapitre 2, nous décrivons l'arbre de tâches de manière hiérarchique puis nous ajoutons les opérateurs temporels entre les tâches. Nous décrivons ainsi les activités réalisables avec l'application ainsi que l'enchaînement de ces sous-activités nécessaires pour atteindre son but.

### 3.3.1 Structure de l'arbre de tâches

Comme son nom l'indique, ce modèle met en place une structure hiérarchique. Nous considérons alors deux fonctions permettant d'obtenir la tâche parente à une tâche donnée et d'obtenir les sous-tâches d'une tâche donnée.

Soit $TASKTREE$, l'arbre de tâches associé à une application. $TASKTREE = \{TASK_i\}$ ensemble de toutes les tâches constituants l'arbre de tâches.
Soit $isSubtaskOf$ la fonction permettant de décrire qu'une tâche est sous-tâche d'une autre, $subtasks$ la fonction permettant de récupérer toutes les sous-tâches d'une tâche donnée et $parentOf$ la fonction permettant de récupérer la tâche parente à une tâche donnée :

- $isSubtaskOf : TASKTREE \times TASKTREE \rightarrow BOOLEAN$
  $\forall t_i, t_j \in TASKTREE$,
  $isSubtaskOf(t_i, t_j) \Leftrightarrow t_j$ est une sous-tâche de $t_i$
- $subtasks : TASKTREE \rightarrow \mathcal{P}(TASKTREE)$
  $\forall t_i \in TASKTREE$,
  $subtasks(t_i) = \{t_j\} / \forall j, t_i \neq t_j \wedge isSubtaskOf(t_j, t_j)$
- $parentOf : TASKTREE \rightarrow TASKTREE$
  $\begin{cases} \forall t_i \in TASKTREE, \\ parentOf(t_i) = t_j \Leftrightarrow t_i \in subtasks(t_j) \\ parentOf(t_i) = \varnothing \text{ si } t_i \text{ n'a pas de tâche parente, } i.e., \text{ c'est la tâche racine} \end{cases}$

Nous définissons également une fonction pour savoir si une tâche est l'ancêtre d'une autre :

$isTaskAncestorOf(task_{parent}, task_{child}) : TASKTREE \times TASKTREE \rightarrow BOOLEAN$
$\forall task_{parent}, task_{child} \in TASKTREE$,
$isTaskAncestorOf(task_{parent}, task_{child}) \Leftrightarrow (task_{parent} = task_{child}) \vee (task_{parent} = parentOf(task_{child})) \vee (\exists task_{path} \in TASKTREE / task_{path} \in subtasks(task_{parent}) \wedge isTaskAncestorOf(task_{path}, task_{child}))$

Nous obtenons alors la description hiérarchique de l'arbre de tâches d'une application comme l'arbre correspondant à l'application "Maps" représenté sur la figure 3.8. Par exemple, la tâche "Fill begin and arrival positions" correspond à l'action de renseigner la position de départ et celle d'arrivée du futur itinéraire. Cette tâche est composée de deux sous tâches : "Fill Position A", *i.e.*, saisir la position de départ, et "Fill Position B", *i.e.*, saisir la position d'arrivée.

### 3.3.2 Types de tâches

A la manière de CTT [34], nous définissions quatre types de tâches. Il y a :

- des tâches utilisateurs (*User*), représentant les activités devant être réalisées par l'utilisateur (activités cognitives),
- des tâches systèmes (*System*), représentant les activités réalisées par le système de l'application (appel d'un service, recherche de données, ...),
- des tâches d'interactions (*Interaction*), représentant les activités d'interactions entre l'utilisateur et l'application (l'utilisateur devant agir sur l'application, l'application faisant un retour à l'utilisateur, ...) et
- des tâches abstraites (*Abstract*) constituées de sous-tâches de types différents.

Par exemple, sur la figure 3.8, les tâches "Fill Position A" et "Fill Position B" sont des tâches d'interactions car elles correspondent à des saisies de texte (adresse) par l'utilisateur. "Fill begin and arrival positions" est également une tâche d'intéraction car elle est composée uniquement de tâches du même type. "Route Calcul" est une tâche système qui correspond au calcul de l'itinéraire. La tâche "Retrieve a Route" est donc une tâche abstraite car elle est composée, entre autre, d'une tâche d'intéraction ("Fill begin and arrival positions") et d'une tâche système ("Route Calcul").

Nous définissons formellement ces types de tâches :

Soit $TASKSTYPES = \{ABSTRACT, USER, SYSTEM, INTERACTION\}$, l'ensemble des types possibles pour une tâche.
En considérant :
$typeTask : TASKTREE \rightarrow TASKSTYPES$ la fonction permettant de récupérer le type de la tâche, nous obtenons 4 fonctions permettant de valider le type d'une tâche :

- Une fonction qui permet de savoir si la tâche est de type ABSTRACT
  $isAbstract : TASKTREE \rightarrow BOOLEAN$ tel que
  $\forall t \in TASKTREE, isAbstract(t) \Leftrightarrow typeTask(t) = ABSTRACT$
- Une fonction qui permet de savoir si la tâche est de type USER
  $isUser : TASKTREE \rightarrow BOOLEAN$ tel que
  $\forall t \in TASKTREE, isUser(t) \Leftrightarrow typeTask(t) = USER$
- Une fonction qui permet de savoir si la tâche est de type SYSTEM
  $isSystem : TASKTREE \rightarrow BOOLEAN$ tel que
  $\forall t \in TASKTREE, isSystem(t) \Leftrightarrow typeTask(t) = SYSTEM$
- Une fonction qui permet de savoir si la tâche est de type INTERACTION
  $isInteraction : TASKTREE \rightarrow BOOLEAN$ tel que
  $\forall t \in TASKTREE, isInteraction(t) \Leftrightarrow typeTask(t) = INTERACTION$

Nous définissons alors une propriété sur les tâches mettant en évidence que la tâche ne peut être que d'un seul type :

$\forall t \in TASKTREE,$
- $isAbstract(t) = \neg isUser(t) \wedge \neg isSystem(t) \wedge \neg isInteraction(t)$
- $isUser(t) = \neg isSystem(t) \wedge \neg isInteraction(t) \wedge \neg isAbstract(t)$
- $isSystem(t) = \neg isInteraction(t) \wedge \neg isAbstract(t) \wedge \neg isUser(t)$
- $isInteraction(t) = \neg isAbstract(t) \wedge \neg isUser(t) \wedge \neg isSystem(t)$

### 3.3.3 Relations temporelles entre tâches

A la description hiérarchique de l'arbre de tâches, nous ajoutons les relations temporelles entre les tâches définies dans CTT [34] et dont les différents opérateurs temporels sont visibles sur la figure 2.1 (c.f. page 22). Ces relations temporelles permettent de décrire l'enchaînement entre les tâches pour remplir une tâche de plus haut niveau.

N'ayant pas eu besoin d'exprimer les relations entre les tâches de manière aussi précise que les opérateurs LOTOS [28] utilisés dans CTT, nous ne représentons ici que quatre relations : les tâches à faire sans ordre imposé (*isInParrallelWith*), les tâches à faire dans un ordre imposé avec un transfert d'information entre les tâches (*isInSequenceWith*), un choix parmi plusieurs tâches (*isInAChoiceWith*) et l'activation d'une tâche par une autre, i.e., une séquence mais sans transfert d'information (*enablingNext*). Cet ensemble de relations temporelles,
noté $TEMPOP = \{isInParrallelWith, isInSequenceWith, isInAChoiceWith, enablingNext\}$, est facilement extensible. Nous définissons les relations temporelles de la manière suivante :

*getTempOpBetween* permet de définir la relation temporelle présente entre deux tâches données :
$getTempOpBetween : TASKTREE \times TASKTREE \rightarrow TEMPOP$
$\forall t_i, t_j \in TASKTREE$,
$getTempOpBetween(t_i, t_j) = tempop, tempop \in TEMPOP \Leftrightarrow t_i \neq t_j \wedge \exists tempop \in TEMPOP / tempop(t_i, t_j)$ c'est-à-dire qu'il existe une relation temporelle entre $t_i$ et $t_j$. Si aucune relation temporelle entre les deux tâches n'existe alors :
$getTempOpBetween(t_i, t_j) = \varnothing$

Les fonctions suivantes définissent les relations temporelles.

- $isInParrallelWith : TASKTREE \times TASKTREE \rightarrow BOOLEAN$
  $\forall t_i, t_j \in TASKTREE$,
  $isInParrallelWith(t_i, t_j) \Leftrightarrow t_j$ est une tâche en parallèle de la tâche $t_i$
- $isInSequenceWith : TASKTREE \times TASKTREE \rightarrow BOOLEAN$
  $\forall t_i, t_j \in TASKTREE$,
  $isInSequenceWith(t_i, t_j) \Leftrightarrow t_j$ est une tâche en séquence de la tâche $t_i$ ( $t_j$ s'exécutant à la suite de $t_i$ )
- $isInAChoiceWith : TASKTREE \times TASKTREE \rightarrow BOOLEAN$
  $\forall t_i, t_j \in TASKTREE$,
  $isInAChoiceWith(t_i, t_j) \Leftrightarrow t_j$ est une tâche en choix avec la tâche $t_i$
- $enablingNext : TASKTREE \times TASKTREE \rightarrow BOOLEAN$
  $\forall t_i, t_j \in TASKTREE$,
  $enablingNext(t_i, t_j) \Leftrightarrow t_i$ est une tâche actionnant la tâche $t_j$

Avec la représentation des opérateurs présents dans CTT, nous notons ces relations temporelles sur l'arbre de tâches associé à l'application "Maps" sur la figure 3.8. Par exemple les tâches "Fill Position A" et "Fill Position B" peuvent être fait dans n'importe quel ordre. Dans CTT, le terme utilisé est *Independant Concurrency*. Dans le formalisme ci-dessus, nous noterons cette relation *isInParrallelWith*. En revanche, la tâche "Fill begin and arrival positions" doit être complétée avant de faire "Trigger Route Calcul", *i.e.*, lancer le calcul de l'itinéraire en cliquant sur le bouton. Par ailleurs, il y a un transfert d'information entre ces deux tâches : les adresses saisies par l'utilisateur. Dans CTT, le terme utilisé est *Enabling with information passing*. Dans le formalisme ci-dessus, nous noterons cette relation *isInSequenceWith*.

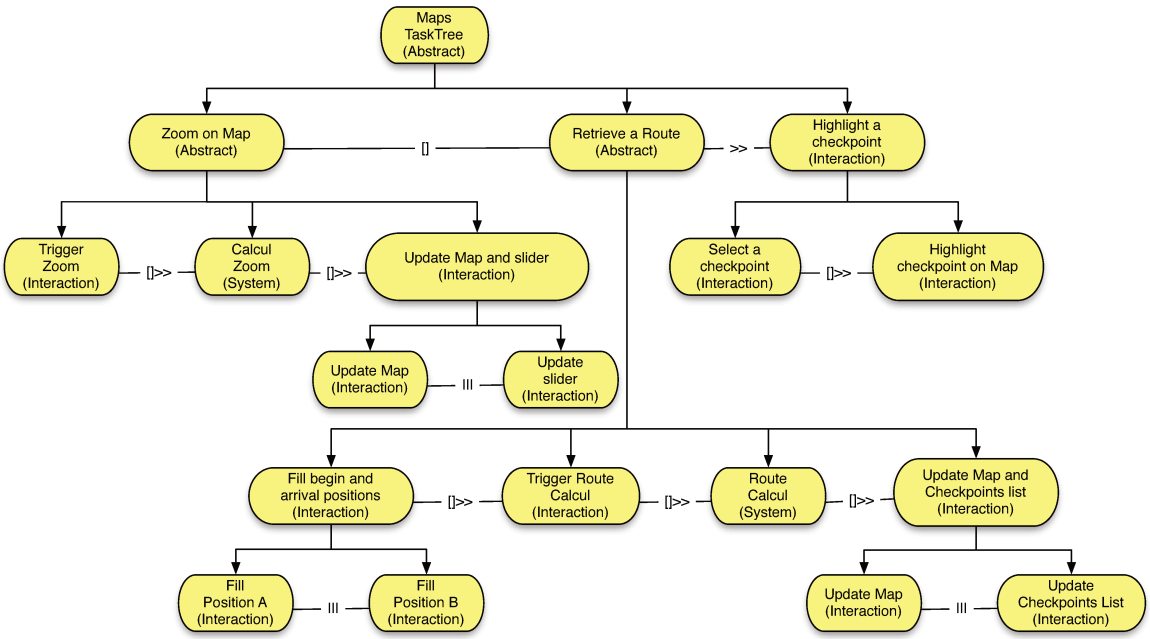



FIGURE 3.8 – Représentation de l'arbre de tâches associé à l'application "Maps".

## 3.4 Liens entre les différents modèles

Nous venons donc de définir une application selon 3 modèles complémentaires, ces trois modèles étant une synthèse de concepts manipulés dans les travaux antérieurs étudiés sur la composition d'applications. Ces modèles portent sur les éléments logiciels constituant une application, les éléments graphiques constituant son interface graphique et les tâches qu'elle permet de réaliser. Ces concepts se recoupent : les éléments logiciels englobent (mais selon un découpage différent) les éléments graphiques et les tâches sont atteintes à travers l'exécution et la manipulation des deux autres concepts.

Notre contribution centrale de notre modèle consiste à expliciter ses liens, ce qui nous permettra de réaliser des compositions d'applications dans leurs intégralités. Nous proposons donc des connexions entre ces modèles deux à deux. Les premiers liens que nous définissons vont être entre le modèle de l'interface graphique de l'application et le modèle opérationnel. Puis nous proposerons d'ajouter les liens entre le modèle de tâches et le modèle opérationnel. Enfin, nous étudierons l'ajout de lien entre le modèle de l'interface graphique et le modèle de tâches.

### 3.4.1 Liens entre le modèle de l'interface graphique et modèle opérationnel

Afin de définir des ponts entre l'interface graphique et les éléments logiciels de l'application, nous lions les éléments graphiques aux ports "UI" ou "UI Component" des éléments logiciels. Le lien avec les ports "UI Component" permettront d'identifier les éléments graphiques "sélectionnables" (cf. chapitre 4) c'est-à-dire qui seront réellement manipulables pendant la composition. Pour cela, il faudra pouvoir les extraire à travers le modèle opérationnel et donc nous avons besoin de savoir quel élément logiciel "fournit" quel(s) élément(s) graphique(s). Les liens avec les ports "UI" permettent de faire le lien entre ces ports spécifiques, qui permettent une interaction avec l'utilisateur et l'élément graphique permettant effectivement cette interaction. Nous définissons de tels liens :

Soit $uiElemLinkedWith$ la fonction permettant de lier un élément graphique à un port fourni de type "UI" : $uiElemLinkedWith : PORTS \times APPUI \rightarrow BOOLEAN$
$\forall p_i \in PORTS, \forall uie_j \in APPUI,$
$uiElemLinkedWith(p_i, uie_j) \Leftrightarrow isProvided(p_i) \wedge (isUIPort(p_i) \vee isUIComponentPort(p_i)) \wedge$
$uie_j$ est un élément d'interface manipulé logiciellement via le port $p_i$

Ainsi, nous définissons la fonction $portsFromUI$ permettant de récupérer tous les ports associés à un élément graphique et la fonction $uiFromPort$ permettant de récupérer tous les éléments graphiques reliés à un port :

$$portsFromUI : APPUI \rightarrow \mathcal{P}(PORTS)$$
$$\forall uie \in APPUI, portsFromUI(uie) = \{p_j \in PORTS, uiElemLinkedWith(p_j, uie)\}$$

$$uiFromPort : PORTS \rightarrow \mathcal{P}(APPUI)$$
$$\forall port \in PORTS, uiFromPort(port) = \{uie_j \in APPUI, uiElemLinkedWith(port, uie_j)\}$$

Nous définissons aussi à partir des fonctions précédentes des fonctions permettant de lier directement les éléments logiciels et éléments graphiques.

Nous définissons donc $appElemFromUI$ pour récupérer l'élément logiciel associé à un élément graphique et la fonction $uiFromAppElem$ permettant de récupérer tous les éléments graphiques liés à un élément logiciel :

$$appElemFromUI : APPUI \rightarrow APPLI$$
$$\forall uie \in APPUI, appElemFromUI(uie) =$$
$$appelem_j \in APPLI / \exists port \in portsFromAppElem(appelem_j), uiElemLinkedWith(port, uie)$$

$$uiFromAppElem : APPLI \rightarrow \mathcal{P}(APPUI)$$
$$\forall appelem \in APPLI, uiFromAppElem(appelem) =$$
$$\{uie_j \in APPUI, \exists port \in portsFromAppElem(appelem) / uiElemLinkedWith(port, uie_j)\}$$

Avec ces fonctions, nous lions les éléments graphiques aux éléments logiciels de l'application, comme par exemple pour l'application "Maps" (cf. figure 3.9). Par exemple, le port "UI" de l'élément logiciel "AddressBInput" relié avec "RouteCalculController" est relié à l'élement graphique "AddressBInput" qui permet de saisir la destination. Ce lien représente le fait que "RouteCalculController" a besoin d'obtenir une adresse ici saisie par l'utilisateur.

Autre exemple, le port "UI Component" de l'élément logiciel "AddressBInput" relié avec "PositionsContainer" est associé au même élément graphique "AddressBinput". En effet, ce port permet de fournir cet élément afin qu'il soit ajouté dans la structure hiérarchique de l'interface graphique.

Enfin nous illustrons l'identification des éléments graphiques "sélectionnables". Dans cet exemple de l'application "Maps", qui correspond à une implémentation possible, le *label* "Label 'Position B'" associé au champ de texte pour saisir l'adresse n'est relié directement à aucun élément logiciel. Cela signifie que cet élément graphique est porté par son élément graphique parent, ici "PositionsContainer". Celui-ci est relié à l'élément logiciel du même nom, ce qui implique que l'élément graphique "Label 'Position B'" est encapsulé dans l'élément logiciel "PositionsContainer". Il n'est donc pas possible (sans modification du code source) d'extraire uniquement cet élément graphique de l'application. Finalement l'élément logiciel "AddressBInput" est relié à un seul élément graphique (l'élément graphique "AddressBInput"). Ce dernier ne contient aucun autre élément graphique. Dans ce cas-là, il y

a une sorte d'identité entre les deux éléments. Il faut toutefois noter que l'élément logiciel "Address-BInput" peut aussi englober des traitements logiciels et qu'il "filtre" via le reste de l'application les données et événements en provenance de l'élément graphique "AddressBInput".

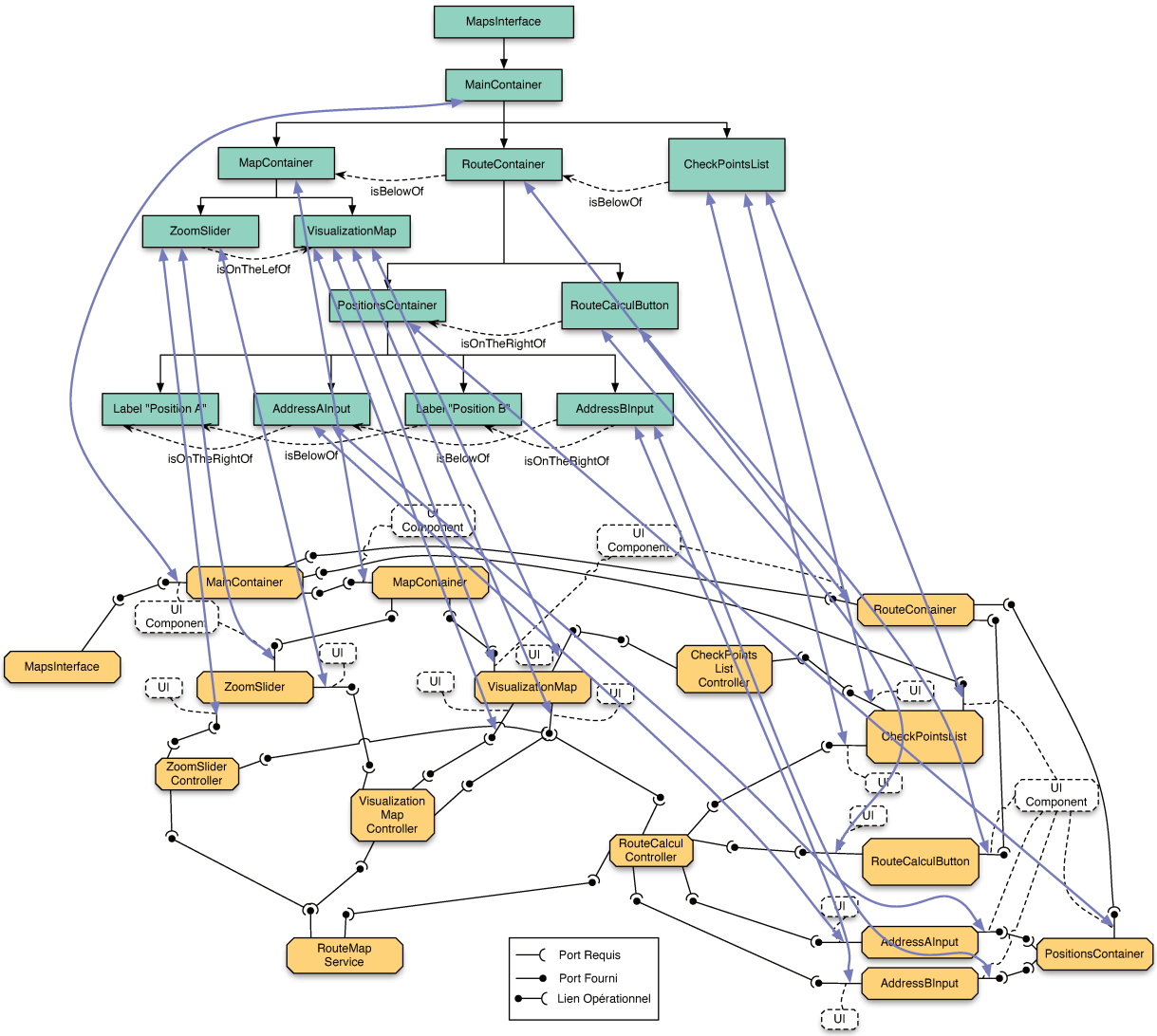



FIGURE 3.9 – Représentation des liens entre les éléments graphiques et les ports des éléments logiciels de l'application "Maps".

### 3.4.2 Liens entre le modèle de tâches et le modèle opérationnel

Il existe un lien évident entre les besoins utilisateurs et l'application. Ces besoins étant modéliser sous forme d'un arbre de tâches, il est alors possible de tisser des liens entre ce même modèle de tâches et le modèle opérationnel (*i.e.*, le découpage en éléments logiciels). Plus précisément, nous mettons en place des liens entre les tâches et les ports des éléments de l'application. Effectivement, les tâches présentes dans l'arbre peuvent être des tâches d'interactions avec l'utilisateurs ou des tâches systèmes, voire des tâches abstraites combinant les deux types de tâches précédents. Ainsi, nous lions une tâche

à un ou plusieurs ports d'un ou de plusieurs éléments logiciels de l'application. Nous identifions deux types de relations :

- les liens entre les tâches d'interactions (ou abstraites) et les éléments logiciels englobant les éléments graphiques correspondants et
- les liens entre les tâches systèmes (ou abstraites) et les éléments logiciels réalisant, entre autre, les fonctionnalités correspondantes.

Afin de définir des ponts entre l'arbre de tâches et les éléments de l'application, nous considérons la fonction suivante :

$$taskLinkedWith : PORTS \times TASKTREE \rightarrow BOOLEAN$$
$$\forall p_j \in PORTS, \forall t_i \in TASKTREE,$$
$$taskLinkedWith(p_j, t_i) \Leftrightarrow t_i \text{ est une tâche liée au port } p_j$$

Ainsi, nous définissons la fonction *portsFromTask* permettant d'identifier tous les ports associés à une tâche, et *tasksFromPort* permettant de récupérer toutes les tâches associées à un port d'un élément logiciel de l'application :

$$portsFromTask : TASKTREE \rightarrow \mathcal{P}(PORTS)$$
$$\forall task \in TASKTREE, portsFromTask(task) = \{p_j \in PORTS, taskLinkedWith(p_j, task)\}$$

$$tasksFromPort : PORTS \rightarrow \mathcal{P}(TASKTREE)$$
$$\forall port \in PORTS, tasksFromPort(port) = \{task_j \in TASKTREE, taskLinkedWith(port, task_j)\}$$

Nous définissons aussi à partir des fonctions précédentes des fonctions permettant de lier directement les éléments logiciels et les tâches.

Nous définissons donc *appElemFromTask* pour récupérer tous les éléments logiciels associés à une tâche et la fonction *taskFromAppElem* permettant de récupérer toutes les tâches liées à un élément logiciel :

$$appElemFromTask : TASKTREE \rightarrow \mathcal{P}(APPLI)$$
$$\forall task \in TASKTREE, appElemFromTask(task) =$$
$$\{appelem_j \in APPLI, \exists port \in portsFromAppElem(appelem_j) / taskLinkedWith(port, task)\}$$

$$taskFromAppElem : APPLI \rightarrow \mathcal{P}(TASKTREE)$$
$$\forall appelem \in APPLI, taskFromAppElem(appelem) =$$
$$\{task \in TASKTREE, \exists port \in portsFromAppElem(appelem), / taskLinkedWith(port, task)\}$$

Nous lions ainsi les tâches aux ports des éléments logiciels de l'application, comme l'illustre la figure 3.10 pour l'application "Maps". Nous en tirons plusieurs exemples :

- La tâche "Route Calcul" (tâche système) est reliée à l'un des ports de l'élément logiciel "RouteMap Service" qui correspond au calcul d'itinéraire.
- La tâche "Fill Position B" (tâche d'interaction de saisie de l'adresse de destination) est reliée au port "UI" de l'élément logiciel "AddressInputB"
- Une tâche peut être reliée à plusieurs éléments logiciels comme l'illustre les liens entre la tâche "Trigger Zoom" (tâche d'interaction déclenchant un zoom avant ou arrière sur la carte) et les éléments logiciels "ZoomSlider" et "VisualizationMap". Pour réaliser le zoom, il y a deux possibilités : soit utiliser le *slider* pour définir le niveau de zoom, soit interagir avec la carte (raccourcis claviers, molette de la souris, etc.).

– Un élément logiciel peut intervenir dans plusieurs tâches, comme le montre l'élément logiciel "VisualizationMap". Il peut en effet déclencher un zoom (lien avec la tâche "Trigger Zoom"), mettre en valeur un point de passage (lien avec la tâche "Highlight checkpoint on Map") et afficher la carte (après un zoom ou après un calcul d'itinéraire, liens avec les tâches nommées "Update Map").

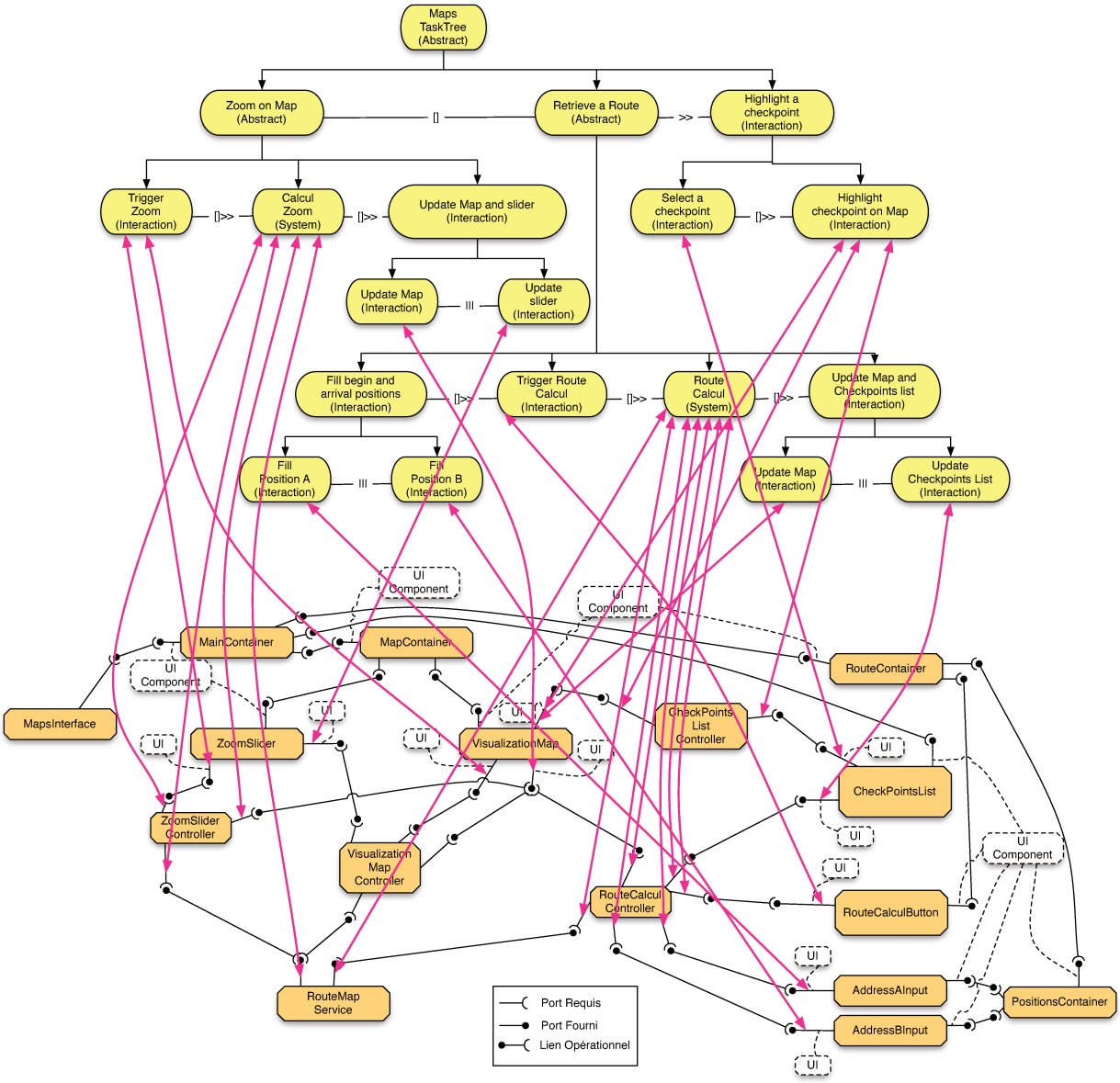



FIGURE 3.10 – Représentation des liens entre les tâches et les ports des éléments logiciels de l'application "Maps".

### 3.4.3 Liens entre modèle de l'interface graphique et l'arbre de tâches

Les tâches d'interactions permettent de décrire une activité qui nécessite une interaction entre l'utilisateur et l'application (interaction provenant de l'utilisateur ou provenant de l'application). Nous

mettons en place des liens qui permettent de lier ces tâches d'interactions avec les éléments graphiques permettant cette interaction. Ce lien est créé entre un élément graphique et une tâche de type $INTERACTION$ ou $ABSTRAITE$.

Nous considérons donc les fonctions suivantes permettant de mettre en place de tels liens.

Soit $linkUIElemTask$ la fonction permettant de lier un élément graphique à une tâche de type $INTERACTION$

$linkUIElemTask : TASKTREE \times APPUI \rightarrow BOOLEAN$
$\forall t_i \in TASKTREE, \forall uie_j \in APPUI,$
$linkUIElemTask(t_i, uie_j) \Leftrightarrow (isInteraction(t_i) \vee isAbstract(t_i)) \wedge uie_j$ est un élément d'interface utilisé pour réalisera tâche $t_i$

Ainsi, nous définissons la fonction $tasksFromUI$ permettant de récupérer toutes les tâches associées à un élément graphique et la fonction $uiFromTask$ permettant de récupérer tous les éléments graphiques reliés à une tâche :

$tasksFromUI : APPUI \rightarrow \mathcal{P}(TASKTREE)$
$\forall uie \in APPUI, tasksFromUI(uie) = \{t_j \in TASKTREE, linkUIElemTask(t_j, uie)\}$

$uiFromTask : TASKTREE \rightarrow \mathcal{P}(APPUI)$
$\forall task \in TASKTREE, uiFromTask(task) = \{uie_j \in APPUI, linkUIElemTask(task, uie_j)\}$

Avec ces fonctions, nous lions les tâches aux éléments graphiques de l'application, comme par exemple pour l'application "Maps" (cf. figure 3.11, page 60). Par exemple la tâche "Trigger Zoom" est reliée aux éléments graphiques "ZoomSlider" et "VisualizationMap". Ou encore l'élément graphique "VisualizationMap" est relié à quatre tâches : "Trigger Zoom", "Highlight checkpoint on Map", "Update Map" (après un zoom) et "Update Map" (après un calcul d'itinéraire).

## 3.5 Conclusion

Dans ce chapitre, nous avons présenté notre modèle d'application orienté vers la composition. Nous basons notre composition sur des éléments logiciels qui forment un découpage de l'application. Pour chaque élément logiciel, les ports requis et les ports fournis sont connus. Cependant nous n'avons aucune hypothèse supplémentaire sur les éléments logiciels, en particulier nous ne supposons pas avoir accès au code source, mais juste aux définitions des ports requis et des ports fournis. Cependant certaines informations présentes de notre modèle doivent être explicitées et disponibles.

Nous modélisons une application selon trois points de vue utilisés généralement séparément dans la composition d'applications (cf. chapitre 2) :

- un modèle opérationnel qui décrit les éléments logiciels et leurs inter-connexions. Ce modèle est inspiré de l'approche à composants. Il permet de différencier les ports liés à l'interface graphique (annotés "UI" ou "UI Component") des autres,
- un modèle de l'interface graphique qui décrit la structure hiérarchique "conteneur-contenu" des éléments graphiques ainsi que leurs relations géométriques et
- un modèle des besoins utilisateurs à travers un arbre de tâches dont les relations entre tâches sont celles de CTT [34].

Notre contribution principale porte sur la connexion explicite entre ces modèles :
- les liens entre le modèle opérationnel et le modèle de l'interface graphique : savoir quel élément logiciel englobe quel(s) élément(s) graphique(s) et réciproquement
- les liens entre le modèle opérationnel et l'arbre de tâches : savoir quels éléments logiciels sont utilisés pour atteindre un objectif (réaliser une tâche) et savoir comment est utilisé un élément logiciel (par quelles tâches)
- les liens entre le modèle de l'interface graphique et l'arbre de tâches : savoir quels éléments graphiques sont utilisés pour atteindre un objectif (réaliser une tâche) et savoir comment est utilisé un élément graphique (par quelles tâches).

Dans les deux prochains chapitres, nous présentons comment utiliser ce modèle. Nous identifions deux actions "élémentaires" pour la composition : il faut pouvoir désigner ce qu'il faut composer, ce qui est présenté dans le chapitre 4 et il faut pouvoir dire comment composer, ce qui est présenté au chapitre 5.

## Publications

Les modèles permettant de représenter les 3 points de vue d'une application ont été présentés au fur et à mesure de leur évolution dans nos publications. Les publications suivantes sont concernées :

### Publications dans des Conférences Internationales

[BPDFZ+11] Christian Brel, Anne-Marie Pinna-Déry, Catherine Faron-Zucker, Philippe Renevier, and Michel Riveill. OntoCompo : An Ontology-Based Interactive System To Compose Applications. In *Seventh International Conference on Web Information Systems and Technologies (WEBIST 2011), short paper*, pages 322–327. Springer-Verlag, May 2011.

[BPDRR11] Christian Brel, Anne-Marie Pinna-Déry, Philippe Renevier, and Michel Riveill. OntoCompo : A Tool To Enhance Application Composition. In *13th IFIP TC13 Conference in Human-Computer Interaction INTERACT 2011 (Interact 2011)*, pages 588–591, September 2011.

[BRO+10] Christian Brel, Philippe Renevier, Audrey Occello, Anne-Marie Pinna-Déry, Catherine Faron-Zucker, and Michel Riveill. Application Composition Driven By UI Composition. In *3rd International Conference on Human Computer Software Engineering (HCSE 2010)*, volume 6409 of *LNCS*, pages 198–205. LNCS, October 2010.

[BRPDR12a] Christian Brel, Philippe Renevier, Anne-Marie Pinna-Déry, and Michel Riveill. Annotated Component-Based Description for Application Composition. In *The Seventh International Conference on Software Engineering Advances (ICSEA 2012)*, November 2012.

[BRPDR12b] Christian Brel, Philippe Renevier, Anne-Marie Pinna-Déry, and Michel Riveill. Application and UI composition using a Component-Based Description and Annotations. In *38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2012)*, pages 204–207, September 2012.

[BRPDR12c] Christian Brel, Philippe Renevier, Anne-Marie Pinna-Déry, and Michel Riveill. UI Modeling as Ontology for Composition. In *The Twenty First International Conference On Software Engineering and Data Engineering (SEDE 2012)*, pages 67–72, June 2012.

## Publications dans des Conférences Nationales

[BM11] Christian Brel and Sébastien Mosser. Vers une approche flot de données pour supporter la composition d'interfaces homme-machine. In *Journées sur l'Ingénierie Dirigée par les Modèles (IDM'11)*, pages 1–7, Lille, June 2011.

[BR11] Christian Brel and Philippe Renevier. Composing Applications with OntoCompo. In *Conference Interaction Homme-Machine (IHM)*, Nice, October 2011.

[Bre09] Christian Brel. Une approche de description d'Interfaces Homme-Machine multi-niveaux. In *MAnifestation des JEunes Chercheurs en Sciences et Technologies de l'Information et de la Communication 2009 (MajecSTIC 2009)*, Avignon, November 2009.

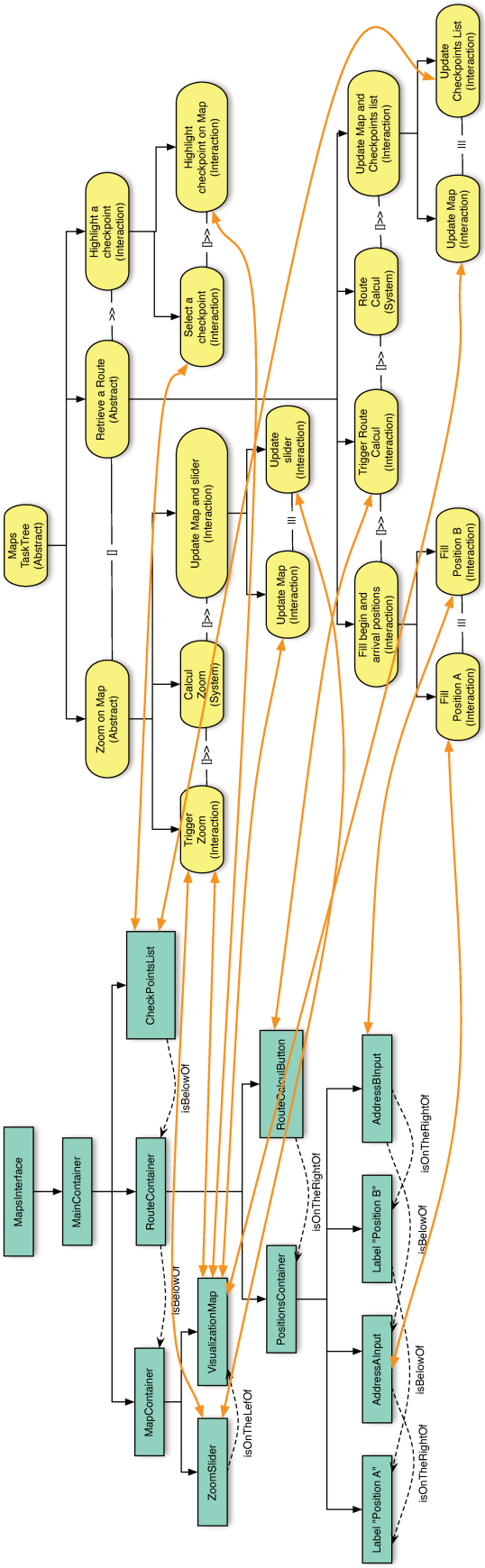



FIGURE 3.11 – Représentation des liens entre les tâches et les éléments graphiques de l'application "Maps".

CHAPITRE 4

# Sélection multi-points de vue de parties d'application

## Sommaire



NOUS présentons dans ce chapitre la notion de sélection et d'extension de sélection. Il s'agit d'une étape inévitable de la composition : il faut pouvoir désigner ce qu'il faut conserver et ce qu'il faut remplacer dans les applications à composer. Nous proposons des moyens pour aider à la sélection, pour en permettre l'enrichissement. Il s'agit donc de fournir des outils à cette désignation, quelle qu'elle soit (humaine ou algorithmique).

Ainsi, une fois les éléments désignés, il est possible d'explorer les liens présents dans le modèle de l'interface graphique, dans le modèle opérationnel et dans le modèle de tâches de l'application. L'exploration de la hiérarchie de l'interface graphique de l'application fournira de l'aide dans la sélection à travers le groupement des éléments de l'interface initiale. L'exploration des liens opérationnels permettra de conserver les éléments nécessaires à l'exécution correcte de l'application. De même avec l'exploration de l'arbre de tâches de l'application, il est possible de conserver les éléments de l'application remplissant une tâche complète. En exprimant les liens entre les modèles, nous permettons des

sélections cohérentes et mettons en évidence les éléments requis manquant.

Dans cette thèse, nous nous intéressons plus particulièrement à la manipulation des interfaces graphiques. Aussi nous axons la présentation de la sélection à travers des définitions propres à la sélection graphique, puis nous explorons à tour de rôle les extensions selon le modèle graphique, selon le modèle opérationnel et selon le modèle de tâches. A chaque fois, les extensions sont répercutées sur les autres modèles. Avant de conclure le chapitre, nous illustrons la combinaison d'extension.

## 4.1 Sélection : définitions et implications

Dans cette section, nous revenons sur la notion d'élément graphique sélectionnable, c'est-à-dire les éléments graphiques qu'il sera possible d'isoler de l'application initiale. Puis nous expliciterons ce qui est "réellement" sélectionné, la sélection d'éléments graphiques impliquant la sélection de tâches ou éléments logiciels, puisque nous ne voulons pas dissocier les points de vue opérationnel, tâche et graphique. Nous présenterons aussi un opérateur pour garantir "l'opérationnalité" d'une extension. Finalement, nous exposerons le principe des extensions présentées dans la section suivante.

### 4.1.1 Notion d'élément graphique sélectionnable

Dans le chapitre précédent, nous avons mis en place les relations entre les modèles définissant une application "composable". Nous faisons la distinction entre les éléments logiciels opérationnels de l'application et les éléments graphiques de l'interface graphique. Effectivement, certains éléments logiciels de l'application peuvent être vus comme une encapsulation directe d'un élément graphique. Mais la plupart du temps, ces éléments logiciels gèrent plusieurs éléments graphiques et exposent à travers leurs ports, uniquement ce qui est nécessaire, masquant le fonctionnement interne. La gestion de ces éléments graphiques est donc complément masquée à l'intérieur de l'élément logiciel "encapsulant".
En revanche, considérons le cas d'un élément graphique, noté $EG_{enfant}$, devant être inséré dans l'interface graphique à l'intérieur d'un autre élément graphique parent, noté $EG_{parent}$. Si $EG_{parent}$ n'est pas géré par le même élément logiciel alors cet élément graphique devra être échangé depuis l'élément logiciel enfant $EG_{enfant}$ vers l'élément logiciel encapsulant $EG_{parent}$. Cette échange se fera par un port fourni par le premier élément logiciel vers un port requis par le second. Le port fourni est alors annoté "UI Component". Les éléments graphiques transitant par de tels ports "UI Component" peuvent être réutilisés individuellement afin d'être mis en place dans la nouvelle application construite par composition.

C'est ainsi que nous définissons les éléments graphiques "sélectionnables". Le meneur de la composition va pouvoir fusionner, ré-agencer, en somme manipuler ces éléments. Dans nos modèles, nous considérons donc un élément graphique "sélectionnable" comme étant une élément graphique ayant un lien avec un port d'un élément logiciel de l'application qui a pour rôle "UI Component" :

$$isUISelectable : APPUI \rightarrow BOOLEAN$$
$$\forall uie \in APPUI, isUISelectable(uie) \Leftrightarrow \exists p \in PORTS/uiElemLinkedWith(p,uie) \wedge isUIComponentPort(p)$$

Concrètement, dans une interface, rien n'indique qu'un élément graphique est "sélectionnable" ou non. Pour cela, l'algorithme 1 (*getSelectableGraphicalElement*) permet de retourner le plus proche

parent "sélectionnable" d'un élément graphique désigné par le développeur. Dans la suite de ce chapitre, pour alléger la notation, nous considérons uniquement les éléments graphiques "sélectionnables" car dans le cas contraire il suffit d'appliquer la fonction *getSelectableGraphicalElement* pour obtenir le "premier" élément graphique "sélectionnable" qui en soit l'ancêtre.

**Algorithme 1** $getSelectableGraphicalElement(uie_{chosen})$

**ENTRÉES:** $uie_{chosen} \in APPUI$.
**SORTIES:** $uie_{selectionable} \in APPUI \cup \{\varnothing\}$.

1: $uie_{selectionable} \leftarrow uie_{chosen}$
2: **tantque** $((uie_{selectionable} \neq \varnothing) \wedge (\neg isUISelectable(uie_{selectionable})))$ **faire**
3: $uie_{selectionable} \leftarrow uiElemParent(uie_{selectionable})$
4: **fin tantque**
5: **retourner** $uie_{selectionable}$

### 4.1.2 Complétion d'une sélection

Nous souhaitons ne pas perdre les liens entre les différents modèles. Pouvoir ne sélectionner que des éléments graphiques serait donc un contre-sens. Aussi, nous définissons la fonction *completionUI* qui permet de réunir tous les éléments logiciels ou tâches reliés aux éléments graphiques déjà sélectionnés :

Nous définissons l'ensemble des sélections possibles : $SELECTION = \mathcal{P}(APPUI) \times \mathcal{P}(APPLI) \times \mathcal{P}(TASKTREE)$
$\forall sel \in SELECTION, sel = (\{uie\}, \{appElem\}, \{task\}) / \{uie\} \in \mathcal{P}(APPUI) \wedge \{appElem\} \in \mathcal{P}(APPLI) \wedge \{task\} \in \mathcal{P}(TASKTREE)$

Alors : $completionUI : \mathcal{P}(APPUI) \rightarrow SELECTION$
$completionUI(\{uie_{chosen}\}) = ($
$\{uie_{chosen}\},$
$\bigcup\{appelem \in APPLI / \exists uie \in \{uie_{chosen}\}, uie \in uiFromAppElem(appelem)\},$
$\bigcup\{task \in TASKTREE / \exists uie \in \{uie_{chosen}\}, uie \in uiFromTask(task)\}$
$)$

Nous créons cette fonction dans le but d'effectuer une sélection "complète". Selon le point d'entrée choisi, il sera possible de sélectionner **directement** que des éléments de ce point de vue (dans notre cas, que des éléments graphiques). Il est donc nécessaire d'ajouter aussi à la sélection les éléments des autres points de vue afin d'augmenter les chances d'obtenir comme résultat de la composition, une application opérationnelle.
En partant d'éléments graphiques, grâce à la fonction ci-dessus (*completionUI*), nous englobons ainsi les tâches et les éléments logiciels reliés à ces éléments graphiques. Nous définissons des fonctions similaires pour chaque point d'entrée différent.

Pour le point d'entrée "Tâches", la fonction *completionTask* est la suivante :

$$completionTask : \mathcal{P}(TASKTREE) \rightarrow SELECTION$$
$$completionTask(\{task_{chosen}\}) = ( \bigcup\{uie \in APPUI / \exists task \in \{task_{chosen}\}, task \in tasksFromUI(uie)\},$$
$$\bigcup\{appelem \in APPLI / \exists task \in \{task_{chosen}\}, task \in taskFromAppElem(appelem)\}, \{task_{chosen}\}$$
$$)$$

Pour le point d'entrée "Éléments logiciels", la fonction *completionAppElem* est définie comme suit :

$$completionAppElem : \mathcal{P}(APPLI) \rightarrow SELECTION$$
$$completionAppElem(\{appElem_{chosen}\}) = ($$
$$\bigcup\{uie \in APPUI / \exists appElem \in \{appElem_{chosen}\} / appElem \in appElemFromUI(uie)\},$$
$$\{appElem_{chosen}\},$$
$$\bigcup\{task \in TASKTREE / \exists appElem \in \{appElem_{chosen}\} / appElem \in appElemFromTask(task)\}$$
$$)$$

### 4.1.3 Consolidation d'une sélection

La consolidation d'une sélection d'éléments logiciels, de tâches et d'éléments graphiques inclue tout ce qui est requis dans le but d'obtenir une extraction de l'application qui soit opérationnelle. Il s'agit de garantir que les éléments compris dans la sélection forme un sous-ensemble opérationnel d'une application. Ainsi, le parcours défini par cette étape dite de "consolidation" se fait sur les éléments logiciels. Il va donc ajouter à la sélection courante les éléments "requis" par ceux déjà sélectionnés pour leur bon fonctionnement. Liés à ces nouveaux éléments logiciels, il peut y avoir des éléments graphiques ou des tâches qui ne sont pas (encore) dans la sélection. Nous ajoutons les éléments nécessaires au bon fonctionnement de la sélection. Nous décidons donc d'ajouter les éléments graphiques puisqu'ils permettent l'opérationnalisation et nous faisons le choix de ne pas ajouter de tâches pour ne pas ajouter de nouvelles fonctionnalités.

Préalablement à la définition de l'algorithme de "consolidation", nous définissons la fonction *getRequiredAppElem* pour déterminer l'ensemble des éléments logiciels requis par un autre élément logiciel $appElem$, *i.e.*, les éléments connectés par un des ports requis de $appElem$ :

$$getRequiredAppElem : APPLI \rightarrow \mathcal{P}(APPLI)$$
$$\forall appElem \in APPLI, getRequiredAppElem(appElem) = \{appElem_{required} \in APPLI / \exists p \in portsFromAppElem(appElem_{required}), \exists q \in portsFromAppElem(appElem), areConnecte(p,q) \wedge isProvided(p) \wedge isRequired(q)\}$$

Nous définissons également la fonction *requiring* pour savoir si, dans une sélection d'éléments logiciels, il reste au moins un port requis connecté à un élément logiciel qui lui-même ne soit pas déjà dans la sélection :

$$requiring : \mathcal{P}(APPLI) \rightarrow BOOLEAN$$
$$\forall selAppElem \in \mathcal{P}(APPLI), requiring(selAppElem) \Leftrightarrow \exists appElem_{ext} \in getRequiredAppElem() / appElem_{requiring} \in selAppElem \wedge appElem_{ext} \notin selAppElem$$

Avec l'algorithme 2, nous définissons la fonction de "consolidation" *consolidate* dont les principales étapes sont :

– la complétion des tâches (*completionTask*) - au cas où celle-ci n'aurait pas été effectuée avant de "consolider" la sélection,
– l'ajout des éléments logiciels requis manquants - comme expliqué ci-dessus - et
– l'ajout des éléments graphiques liés aux nouveaux éléments logiciels.

**Algorithme 2** $consolidate(sel)$

**ENTRÉES:** $sel = (selUie, selAppli, selTask) \in SELECTION/$
$selUie \in \mathcal{P}(APPUI), selAppli \in \mathcal{P}(APPLI), selTask \in \mathcal{P}(TASKTREE)$
**SORTIES:** $sel_{consolidate} = (selUie_{consolidate}, selAppli_{consolidate}, selTask) \in SELECTION/$
$selUie_{consolidate} \in \mathcal{P}(APPUI), selAppli_{consolidate} \in \mathcal{P}(APPLI)$.

$sel_{consolidate} \leftarrow sel \Leftrightarrow (selUie_{consolidate}, selAppli_{consolidate}, selTask) \leftarrow (selUie, selAppli, selTask)$
{complétion des tâches}
$sel_{consolidate} \leftarrow sel_{consolidate} \cup completionTask(selTask)$
$selAppli_{new} \leftarrow \emptyset$
{tant qu'il y a des éléments logiciels dont tous les requis ne sont pas présents dans la sélection}
**tantque** $requiring(selAppli_{consolidate})$ **faire**
  **pour tout** $appElem \in selAppli_{consolidate}$ **faire**
    $addedAppElem \leftarrow getRequiredAppElem(appElem)$
    $selAppli_{new} \leftarrow selAppli_{new} \cup addedAppElem$
  **fin pour**
  $selAppli_{consolidate} \leftarrow selAppli_{consolidate} \cup selAppli_{new}$
**fin tantque**
{recherche des éléments graphiques liés aux nouveaux éléments logiciels}
**pour tout** $appElem \in selAppli_{new}$ **faire**
  $connectedProvidedPorts \leftarrow \{p \in portsFromAppElem(appElem)/\exists app \in selAppli_{consolidate}, \exists q \in portsFromAppElem(app), isProvided(p) \wedge areConnected(p,q)\}$
  $selUie_{new} \leftarrow \{uie \in uiFromAppElem(appElem)/\exists p \in portsFromUI(uie), \exists q \in connectedProvidedPorts, p == q\}$
  $selUie_{consolidate} \leftarrow selUie_{consolidate} \cup selUie_{new}$
**fin pour**
**retourner** $sel_{consolidate}$

### 4.1.4 Principe des extensions

Les outils définis précédemment dans cette section, permettent de manipuler les sélections selon les différents points de vue. En partant d'un élément initial, avec l'une des fonctions *completionAppElem*, *completionUI* ou *completionTask*, il est possible de retrouver un ensemble d'éléments liés entre les trois points de vue. Lorsqu'une sélection est terminée, il est alors nécessaire de la consolider pour être certain de ne pas oublier un élément requis, condition nécessaire à sa réutilisation.

Les sections suivantes présentent comment manipuler et étendre une sélection. Ces extensions seront à faire selon les besoins. L'application "Maps" sert d'illustration tout au long de ces sections.

## 4.2 Extension selon la description de l'interface graphique

La première extension de sélection que nous étudions est l'exploration des liens présents dans la description de l'interface graphique. Cette exploration étend la sélection vers des éléments graphiques proches ou faisant partis d'un même "conteneur". Le modèle de l'interface graphique défini dans la section 3.2 (page 44) contient notamment deux liaisons :

- la relation *uiElemParent* qui permet d'atteindre directement l'élément englobant (parent) d'un autre élément graphique et
- la relation *getPosBetween* qui permet de déterminer les éléments graphiques positionnés relativement à un autre élément graphique.

Soit $uie \in APPUI$, un élément sélectionné de l'interface graphique de l'application. A partir de cet élément, nous décrivons les différentes possibilités d'extension afin d'obtenir un sous-ensemble de l'interface graphique de l'application : par une exploration graphique "englobante" ou par une exploration selon les relations spatiales. Pour ces deux cas, nous présentons d'abord le principe puis nous l'enrichissons en globalisant l'extension aux autres points de vue.

Précisons également que dans une sélection d'éléments graphiques, seuls les éléments graphiques "sélectionnables" doivent être présents.

### 4.2.1 Extension suivant l'élément graphique englobant (parent) : principe

Nous proposons deux types d'extension suivant l'élément graphique englobant afin de permettre une plus grande liberté. La première extension remplace l'élément graphique sélectionné par son ancêtre le plus proche sélectionnable. Cette extension permet de conserver la "mise en page" des éléments présents dans l'élément graphique englobant.

Soit $extUIParentUI$ cette première fonction d'extension.

$$extUIParentUI : APPUI \rightarrow APPUI$$
$$\forall uie \in APPUI, extUIParentUI(uie) =$$
$$\begin{cases} uiElemParent(uie) = \varnothing \Rightarrow \varnothing \\ isUISelectable(uiElemParent(uie)) \Rightarrow uiElemParent(uie) \\ \neg isUISelectable(uiElemParent(uie)) \Rightarrow extUIParentUI(uiElemParent(uie)) \end{cases}$$

La figure 4.1 illustre cette extension appliquée à l'application "Maps" et à partir de la sélection de l'élément "AddressAInput". Cette sélection est remplacée par l'élément "PositionsContainer" contenant les deux "labels" et les deux "inputs". La "mise en page" de ces éléments de formulaire est préservée.

La seconde extension proposée est de se servir de l'élément graphique englobant comme passerelle afin de récupérer les éléments graphiques frères "sélectionnables". Ainsi, à la fin de cette extension, l'élément englobant n'est pas sélectionné mais tous ses éléments fils le sont. Cette extension ne permet donc pas de garder la "mise en page" des éléments présents dans l'élément graphique englobant.

Soit $extUISiblingUI$ cette seconde fonction d'extension.

$$extUISiblingUI : APPUI \rightarrow \mathcal{P}(APPUI)$$
$$\forall uie \in APPUI,$$
$$extUISiblingUI(uie) = \{uie_j \in uiElemChildren(uiElemParent(uie))/isUISelectable(uie_j)\}$$

La figure 4.1 illustre cette extension. A la sélection initiale (élément "AddressAInput") s'ajoute l'élément "AddressBInput" mais les "labels (non "sélectionnables") ne sont pas ajoutés. Cette extension ne garde pas la mise en page, l'élément englobant n'étant pas présent dans la sélection.

Pour conserver les liens avec les autres points de vue, il faut appliquer *completionUI* sur chaque élément graphique de la nouvelle sélection.

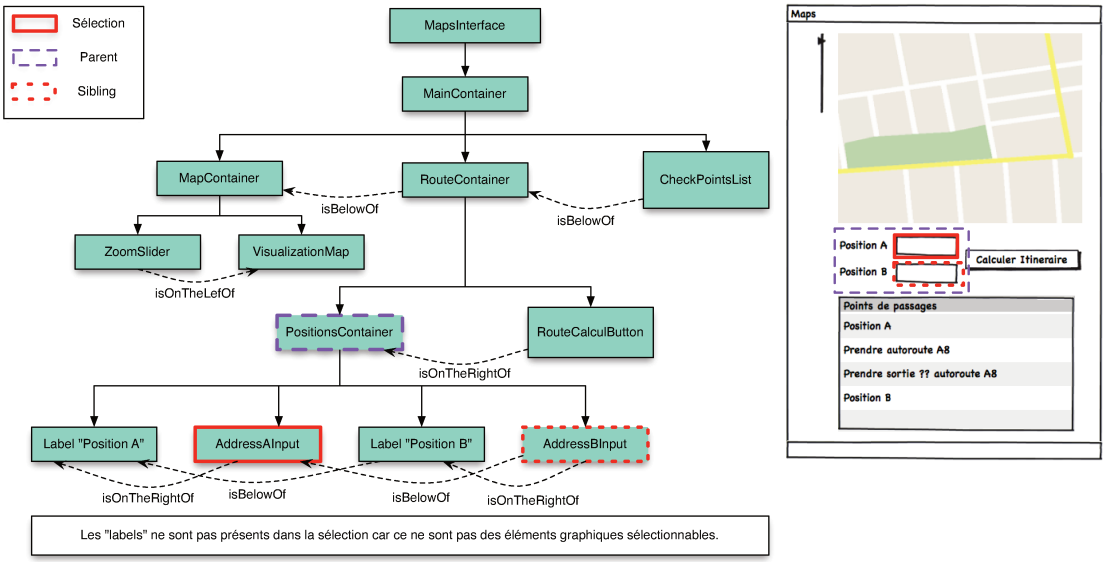



FIGURE 4.1 – Illustration de "l'extension suivant l'élément graphique englobant" appliquée sur l'application "Maps".

### 4.2.2 Extension suivant l'élément graphique englobant (parent) : avec complétion

Nous définissons une fonction d'extension suivant l'élément graphique englobant (parent) intégrant automatiquement la complétion. Soit *extUIParent* et *extUISibling* les deux fonctions intégrants la complétion et faisant écho aux fonctions définies précédemment. Elles sont définies comme suit :

$$
\begin{array}{l}
extUIParent : APPUI \times SELECTION \rightarrow SELECTION \\
\forall uie \in APPUI, \forall sel = (\{uie_{chosen}\}, \{appElem_{chosen}\}, \{task_{chosen}\}) \in \\
SELECTION, extUIParent(uie, sel) = \\
\left\{
\begin{array}{l}
uiElemParentUI(uie) = \varnothing \Rightarrow sel \\
\\
isUISelectable(uiElemParentUI(uie)) \Rightarrow newSel \text{ où} \\
sel_{reduced} \leftarrow sel \setminus (\{uie\}, \emptyset, \emptyset) \text{ et } newSel \leftarrow sel_{reduced} \cup completionUI(\{uiElemParentUI(uie)\}) \\
\\
\neg isUISelectable(uiElemParent(uie)) \Rightarrow extUIParent(uiElemParent(uie), sel)
\end{array}
\right.
\end{array}
$$

$extUISibling : APPUI \times SELECTION \rightarrow SELECTION$
$\forall uie \in APPUI, \forall sel = (\{uie_{chosen}\}, \{appElem_{chosen}\}, \{task_{chosen}\}) \in SELECTION,$
$extUISibling(uie, sel) = sel \cup completionUI(extUISiblingUI(uie))$

La fonction *extUIParent* définit un premier niveau d'exploration de la description de l'interface graphique de l'application.
Nous appliquons ainsi cette fonction de manière récursive pour augmenter le niveau d'exploration de la description de l'interface graphique. Il s'agit de l'appliquer au "parent" de l'élément sélectionné et à la nouvelle sélection pour un second niveau d'exploration (cf. figure 4.2 page 69), puis au "grand-parent" de l'élément sélectionné et la nouvelle nouvelle sélection pour un troisième niveau d'exploration etc... Le premier niveau étant atteint grâce à la fonction ci-dessus, nous définissons les niveaux d'exploration suivants en ajoutant un indice d'exploration :

$extUIParent : APPUI \times SELECTION \times INTEGER \rightarrow SELECTION$
$\forall uie \in APPUI, \forall sel \in SELECTION, \forall deep \in INTEGER \wedge deep \geqslant 0,$
$extUIParent(uie, sel, 0) = sel$ et
$extUIParent(uie, sel, deep) = extUIParent(uiElemParent^{deep-1}(uie), extUIParent(uie, sel, deep - 1))$

- Premier niveau d'exploration : $extUIParent(uie, sel, 1) = extUIParent(uie, sel)$
- Second niveau d'exploration :
  $extUIParent(uie, sel, 2) = extUIParent(uiElemParent(uie), extUIParent(uie, sel, 1))$
- Troisième niveau d'exploration : $extUIParent(uie, sel, 3)$
  $= extUIParent(uiElemParent^2(uie), extUIParent(uie, sel, 2))$
  $= extUIParent(uiElemParent(uiElemParent(uie)), extUIParent(uiElemParent(uie),$
  $extUIParent(uiElemParent(uie), extUIParent(uie, sel, 1)))$
- etc.

### 4.2.3 Extension suivant le positionnement (layout) : principe

Pour effectuer ce nouveau type d'extension de sélection, nous nous basons sur le positionnement des éléments dans l'interface graphique de l'application. A partir d'un élément et d'une "direction" donnée, cette extension permet d'ajouter à la sélection tous les éléments graphiques positionnés dans la "direction" donnée par rapport à l'élément graphique sélectionné (cf. figure 4.3).

Soit *extUIPosUI* cette fonction d'extension suivant une position précise.
$extUIPosUI : APPUI \times POS \rightarrow \mathcal{P}(APPUI)$
$\forall uie \in APPUI, \forall p \in POS, extUIPosUI(uie, p) = \{uie\} \bigcup \{uie_j \in APPUI, /p(uie, uie_j) \wedge isUISelectable(uie)\}$

Nous appliquons *completionUI* sur les éléments graphiques de la nouvelle sélection.

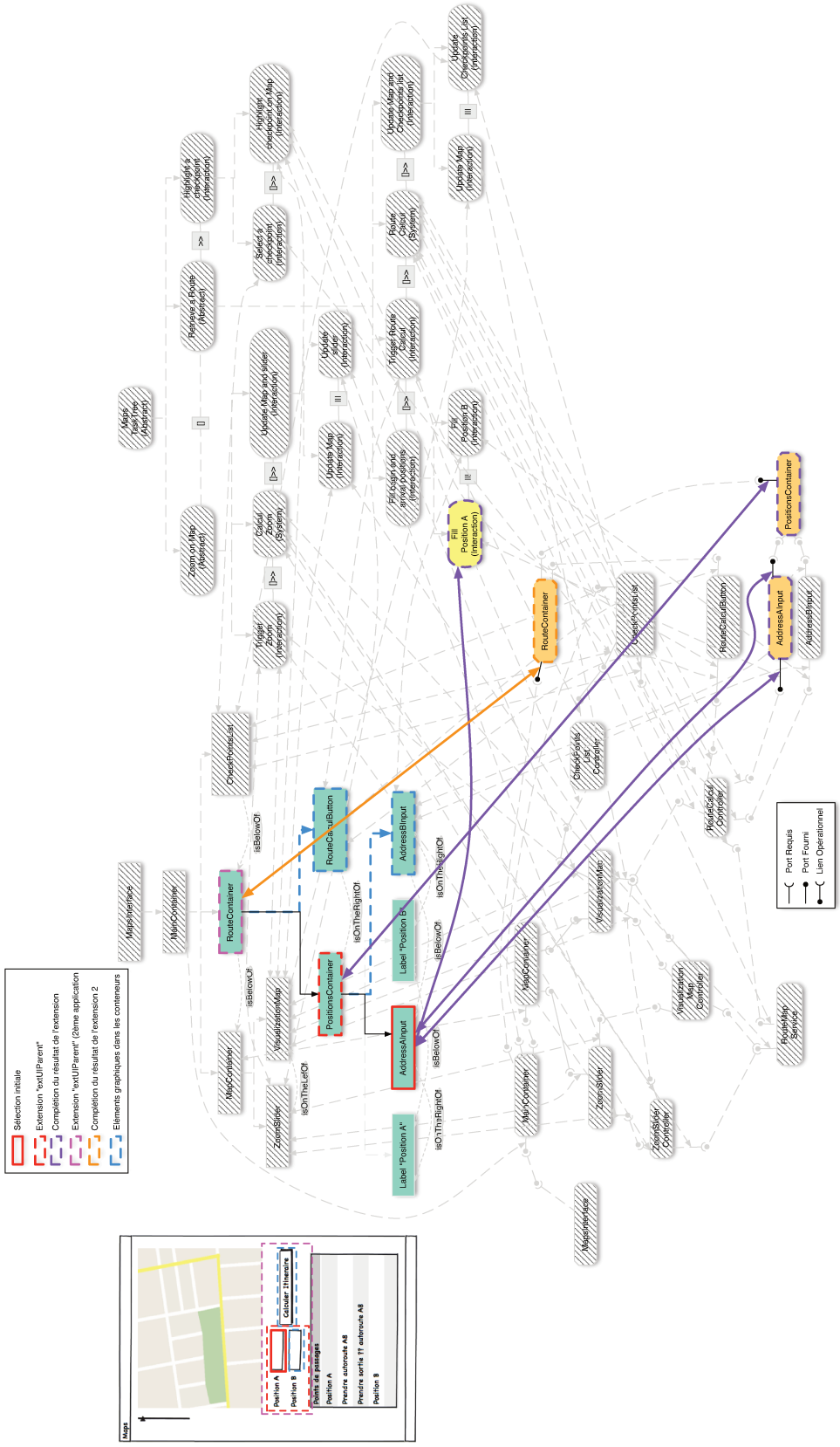



FIGURE 4.2 – Illustration de "l'extension avec complétion suivant l'élément graphique englobant" de niveau 2 appliquée sur l'application "Maps".

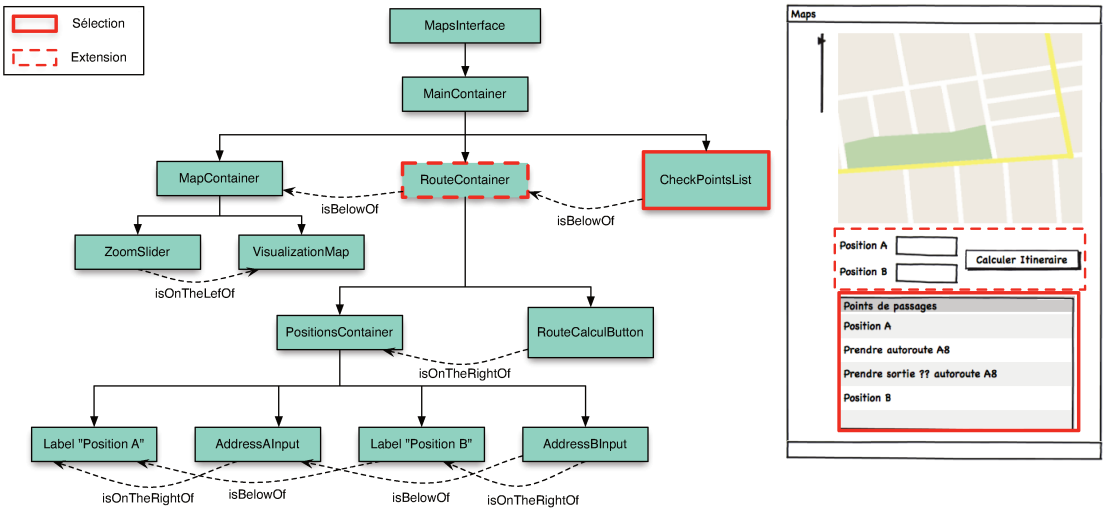



FIGURE 4.3 – Illustration de "l'extension suivant le positionnement (layout)" appliquée sur l'application "Maps".

### 4.2.4 Extension suivant le positionnement (layout) : avec complétion

Nous définissons une fonction d'extension suivant le positionnement (layout) intégrant automatiquement la complétion. Soit *extUIPos* cette fonction :

$$
\begin{array}{l}
extUIPos : APPUI \times POS \times SELECTION \rightarrow SELECTION \\
\forall uie \in APPUI, \forall p \in POS, \forall sel = (\{uie_{chosen}\}, \{appElem_{chosen}\}, \{task_{chosen}\}) \in \\
SELECTION, extUIPos(uie, p, sel) = \\
\begin{cases}
extUIPosUI(uie) = \varnothing \Rightarrow sel \\
\\
extUIPosUI(uie) \neq \varnothing \Rightarrow newSel \text{ où} \\
sel_{completed} \leftarrow \bigcup_{newUie \in extUIPosUI(uie)} \{completionUI(newUie)\} \\
\text{et } newSel \leftarrow sel_{completed} \cup sel
\end{cases}
\end{array}
$$

Cette fonction peut s'appliquer récursivement sur chacun des éléments trouvés. Ainsi, nous pouvons, comme dans le cas de l'extension suivant l'élément graphique englobant, explorer la description de l'interface graphique de l'application, non plus de manière verticale, mais de manière horizontale.

Le premier niveau d'exploration étant définie par la fonction ci-dessus, nous définissons les niveaux suivants en ajoutant la profondeur d'exploration à cette fonction, comme suit :

$$
\begin{array}{l}
extUIPos : APPUI \times POS \times SELECTION \times INTEGER \rightarrow SELECTION \\
\forall uie \in APPUI, \forall p \in POS, \forall sel \in SELECTION, \forall deep \in INTEGER \wedge deep \geqslant 0, \\
extUIPos(uie, p, sel, 0) = sel \text{ et}
\end{array}
$$

$extUIPos(uie, p, sel, deep) = \bigcup_{newUie \in \{uie_{new}\}} \{extUIPos(newUie, p, newSel)\}$ où
$newSel \leftarrow (\{uie_{new}\}, \{appElem_{new}\}, \{task_{new}\}) \in extUIPos(uie, p, sel, deep - 1)$

La figure 4.4 illustre cette extension au niveau 2.

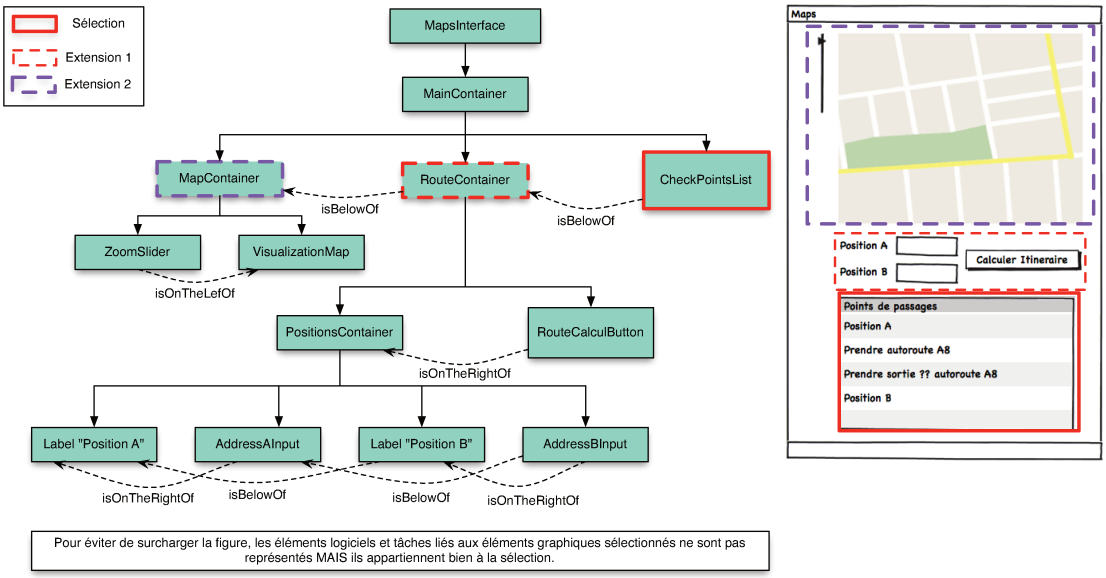



FIGURE 4.4 – Illustration de "l'extension suivant le positionnement (layout)" au niveau 2 appliquée sur l'application "Maps".

Afin de compléter cette fonction d'extension selon une direction, nous proposons de laisser l'opportunité d'explorer le reste de la hiérarchie de l'interface graphique dans la même direction que celle donnée, mais en passant par l'élément englobant parent. Une illustration de cette extension est donnée sur la figure 4.5.

Soit $extUIPosThroughParent$ cette fonction d'extension et $k$ le niveau d'exploration maximisant le nombre d'éléments graphique qu'il est possible de rajouter dans la sélection en utilisant la fonction de base $extUIPos$.

$$extUIPosThroughParent : APPUI \times POS \times SELECTION \times INTEGER \rightarrow SELECTION$$
$$\forall uie \in APPUI, \forall p \in POS, \forall sel \in SELECTION, \forall k \in INTEGER \wedge k > 0,$$
$$extUIPosThroughParent(uie, p, sel, k) = extUIPos(uie, p, sel, k) \bigcup extUIPos(extUIParentUI(uie), p)$$

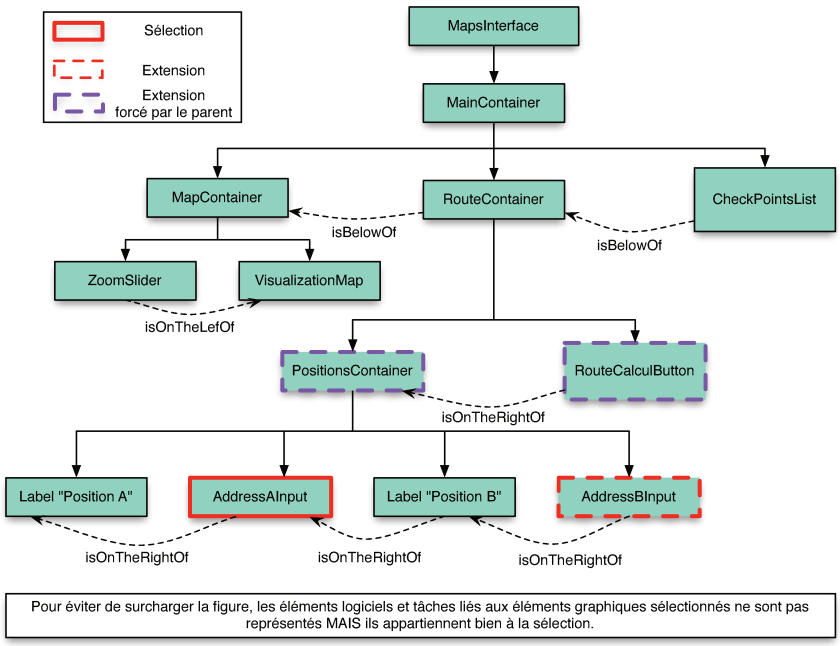



FIGURE 4.5 – Illustration de "l'extension suivant le positionnement (layout)" forcée par l'élément englobant appliquée sur l'application "Maps" (adapté au niveau des positions pour l'illustration).

## 4.3 Extension selon le modèle opérationnel

La seconde extension de sélection que nous étudions est l'exploration des liens présents dans la description du modèle opérationnel. Cette exploration étend la sélection vers des éléments logiciels "proches". Cette proximité est établie par les connexions qu'un élément logiciel peut avoir avec d'autres éléments logiciels.

Soit $appelem \in APPLI$, un élément sélectionné du modèle opérationnel de l'application. A partir de cet élément, nous décrivons les différentes possibilités d'extension afin d'obtenir un sous-ensemble du modèle opérationnel de l'application qui suit les connexions entre éléments logiciels. Nous en présentons d'abord le principe et ensuite nous l'enrichissons en globalisant l'extension aux autres points de vue. Pour finir, nous étudions le cas particulier où cette extension est utilisé dans le cadre d'une composition dirigée par le point de vue "interface graphique", c'est-à-dire comment à partir d'un élément graphique sélectionné atteindre cette extension au niveau modèle opérationnel.

### 4.3.1 Extension suivant les liens opérationnels : principe

Nous explorons les différentes connexions qu'un élément logiciel peut avoir avoir avec d'autres éléments logiciels à travers ses ports (cf. illustration figure 4.6).

Soit $extAppElemApp$ cette fonction d'extension :

$$extAppElemApp : APPLI \rightarrow \mathcal{P}(APPLI),$$
$$\forall appelem \in APPLI, extAppElemApp(appelem) = \{appelem\} \cup \{appelem_j \in APPLI / \exists port_k \in portsFromAppElem(appelem),$$
$$\exists port_m \in portsFromAppElem(appelem_j), areConnected(port_k, port_m)\}$$

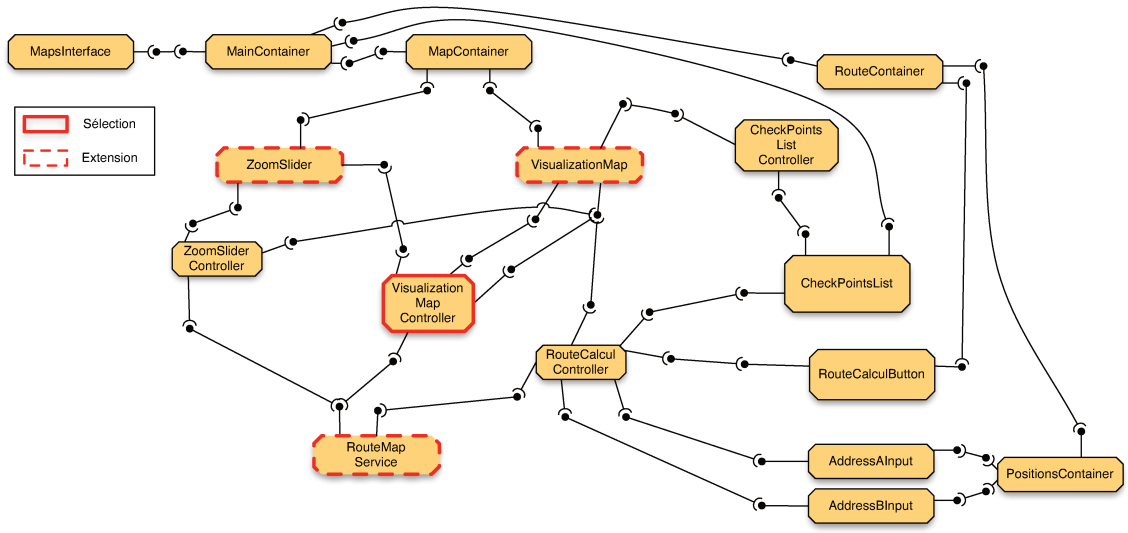



FIGURE 4.6 – Illustration de l'extension suivant les liens opérationnels appliquée sur l'application "Maps".

Pour conserver les liens avec les autres points de vue, il faut appliquer *completionAppElem* sur les éléments logiciels de la nouvelle sélection.

### 4.3.2 Extension suivant les liens opérationnels : avec complétion

Nous définissons une fonction d'extension suivant les liens opérationnels intégrant automatiquement la complétion. Soit *extAppElem* cette fonction définie comme suit :

$$extAppElem : APPLI \times SELECTION \rightarrow SELECTION,$$
$$\forall appelem \in APPLI, \forall sel = (\{uie_{chosen}\}, \{appElem_{chosen}\}, \{task_{chosen}\}) \in SELECTION,$$
$$extAppElem(appelem, sel) =$$
$$\begin{cases} extAppElemApp(appelem) = \varnothing \Rightarrow sel \\ \\ extAppElemApp(appelem) \neq \varnothing \Rightarrow \\ sel \quad \cup \quad completionAppElem(extAppElemApp(appelem)) \end{cases}$$

Cette fonction définit le premier niveau d'exploration de la description des éléments logiciels de l'application. Nous appliquons ainsi cette fonction de manière récursive pour augmenter le niveau d'exploration du modèle opérationnel de l'application. Le premier niveau d'exploration étant atteint grâce à la fonction ci-dessus, nous définissons les niveaux d'exploration suivants en lui ajoutant un indice d'exploration :

$extAppElem : APPLI \times SELECTION \times INTEGER \rightarrow SELECTION$,
$\forall appelem \in APPLI, \forall sel =\in SELECTION, \forall deep \in INTEGER \wedge deep \geqslant 0$,
$extAppElem(appelem, sel, 0) = sel$ et
$extAppElem(appelem, sel, deep) = \bigcup_{newAppelem \in \{appElem_{new}\}} \{extAppElem(newAppelem, newSel)\}$ où
$newSel \leftarrow (\{uie_{new}\}, \{appElem_{new}\}, \{task_{new}\}) \in extAppElem(appelem, sel, deep - 1)$

La figure 4.7 illustre cette extension appliquée à l'application "Maps" à un niveau 2 d'exploration.

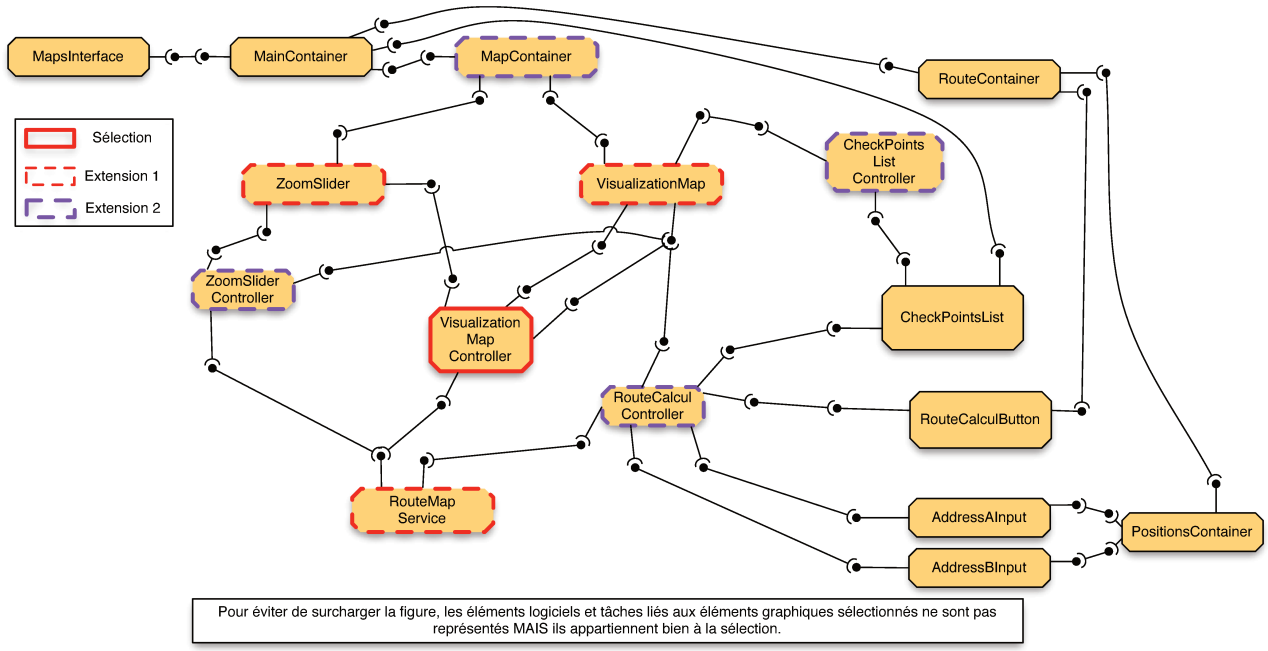



FIGURE 4.7 – Illustration de l'extension suivant les liens opérationnels à un niveau 2 appliquée sur l'application "Maps".

### 4.3.3 Extension suivant les liens opérationnels : à partir d'éléments graphiques

Ayant exploré les connexions entre éléments logiciels de l'application, nous étudions l'application de cette extension "suivant les liens opérationnels", mais en partant d'une sélection d'éléments graphiques.

Dans un premier temps, nous appelons *completionUI* afin de récupérer les éléments logiciels liés aux éléments graphiques sélectionnés. Puis nous appliquons l'extension "suivant les liens opérationnels" avec complétion sur tous les éléments logiciels de la sélection afin d'une part, de récupérer les nouveaux éléments logiciels issus de l'extension, et d'autre part, de remonter à l'interface graphique de l'application en récoltant les éléments graphiques liés aux nouveaux éléments logiciels. Cette fonction $extAppElemFromUI : APPUI \rightarrow SELECTION$ est définie par l'algorithme 3.

**Algorithme 3** $extAppElemFromUI(\{uie_{chosen}\})$

**ENTRÉES:** $\{uie_{chosen}\} \in \mathcal{P}(APPUI)$.
**SORTIES:** $sel = (\{uie_{completed}\}, \{appElem_{completed}\}, \{task_{completed}\}) \in SELECTION$

1: $sel \leftarrow completionUI(\{uie_{chosen}\})$
2: $sel \leftarrow \bigcup_{newAppelem \in \{appElem_{completed}\}} \{extAppElem(newAppelem, sel)\}$
3: $sel \leftarrow sel \bigcup completionAppElem(\{appElem_{completed}\})$
4: **retourner** *sel*

Un exemple de cette extension est illustré par la figure 4.8, à partir de la sélection de l'élément graphique "VisualizationMap". L'appel de la fonction permet de récupérer dans la sélection trois éléments logiciels et un élément graphique supplémentaire.

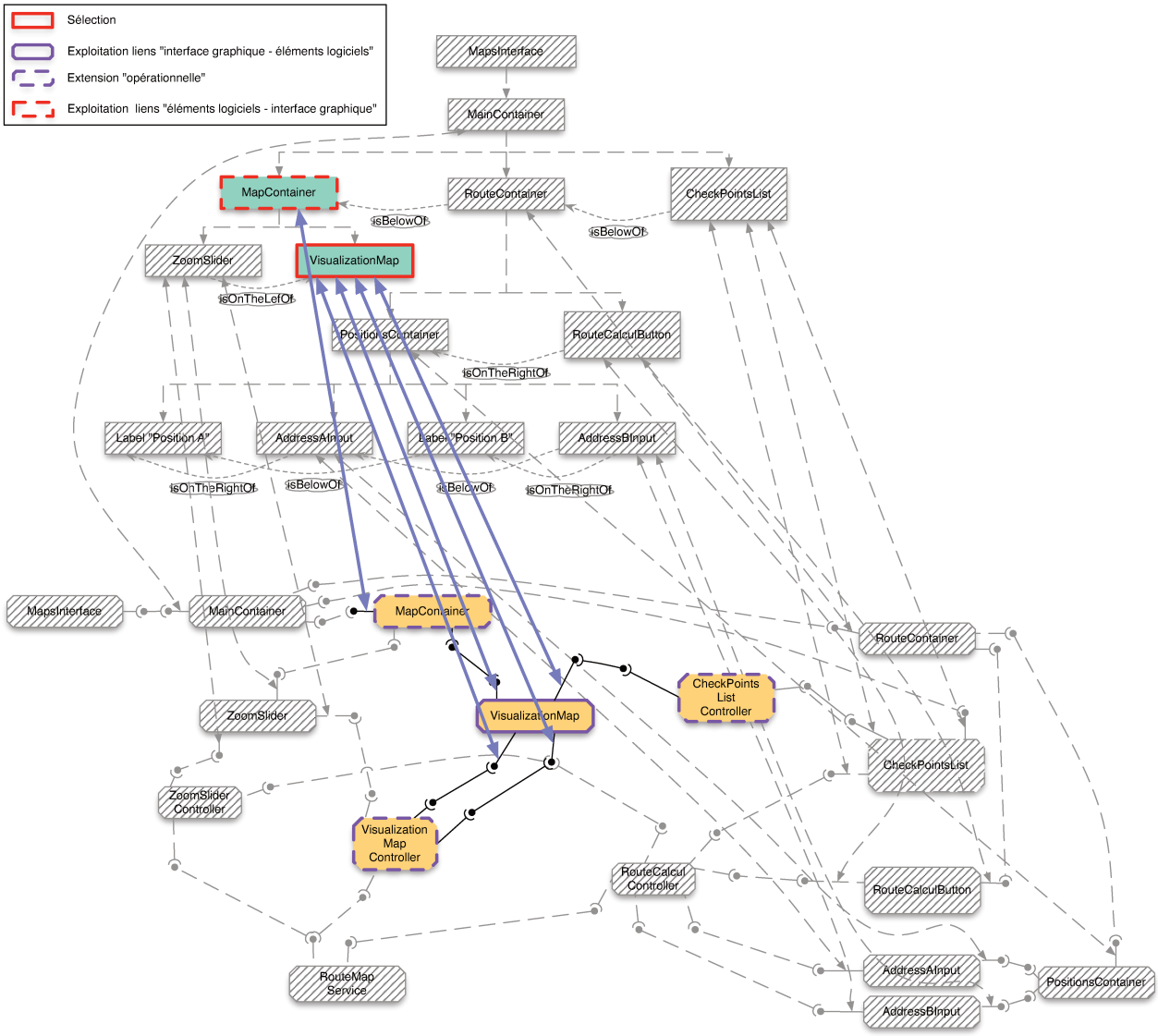



FIGURE 4.8 – Illustration de l'extension "suivant les liens opérationnels" à partir de la sélection d'un élément graphique, appliquée sur l'application "Maps".

## 4.4 Extension selon les besoins

La troisième extension de sélection que nous étudions est l'exploration des liens présents dans la description du modèle de tâches. Cette exploration étend la sélection vers des tâches "proches" ou faisant parties d'une même "tâche parente". Par rapport à la description de l'arbre de tâches établie dans le chapitre 3, à partir d'une tâche, nous atteignons sa tâche parente ou ses tâches enfants ou encore les tâches ayant une relation temporelle par rapport à celle-ci.

Soit $task \in TASKTREE$, une tâche sélectionnée de l'arbre de tâches de l'application. A partir de cette tâche, nous décrivons les différentes possibilités d'extension afin d'obtenir un sous-ensemble de l'arbre de tâches de l'application qui suit l'exploration hiérarchique et temporelle. Pour les deux

cas, nous en présentons d'abord le principe, ensuite nous l'enrichissons en globalisant l'extension aux autres points de vue et finalement nous décrivons l'application de cette extension à partir d'une sélection d'éléments graphiques.

### 4.4.1 Extension suivant la tâche parente : principe

Il s'agit dans cette extension de parcourir la hiérarchie définie par la nature même de l'arbre de tâches de l'application à travers les relations de parentés entre les tâches (cf. figure 4.9).

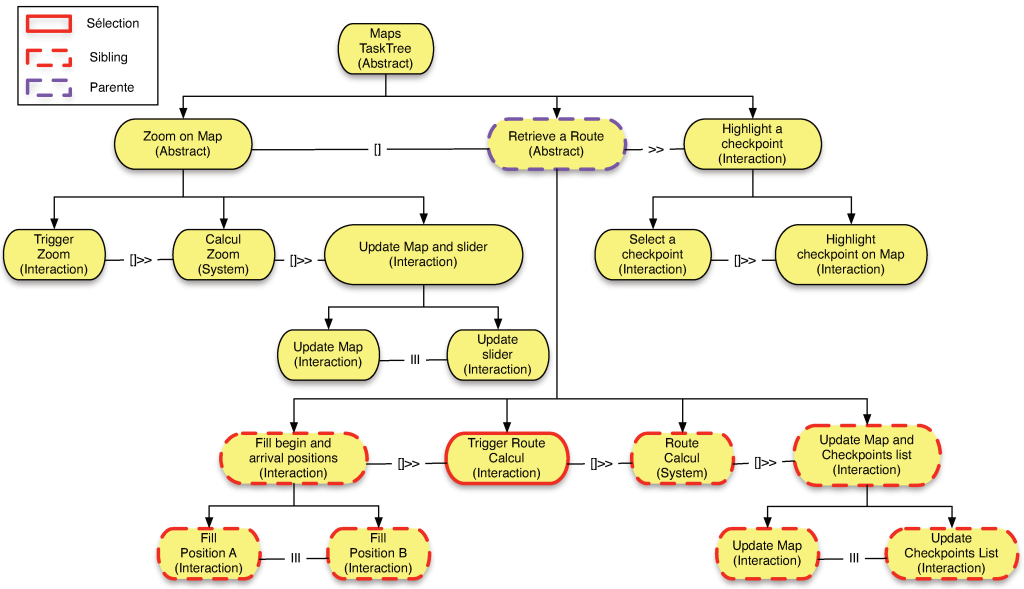



FIGURE 4.9 – Illustration de "l'extension suivant la tâche parente" appliquée sur l'application "Maps".

Nous proposons deux types d'extensions pour cette exploration des liens de parenté afin de permettre une plus grande liberté. La première extension permet de remplacer la tâche sélectionnée par sa tâche parente la plus proche. Cette extension permet de conserver les relations temporelles entre les tâches filles de la tâche parente.

Soit $extTaskParentTask$ cette première fonction d'extension.

$$extTaskParentTask : TASKTREE \rightarrow TASKTREE$$

$$\forall task \in TASKTREE, extTaskParentTask(task) = \begin{cases} parentOf(task) = \varnothing \Rightarrow \varnothing \\ parentOf(task) \neq \varnothing \Rightarrow parentOf(task) \end{cases}$$

La seconde extension proposée est de sélectionner les tâches sœurs en utilisant la tâche parente comme passerelle, celle-ci ne faisant pas partie de la sélection finale. Cette extension ne conserve pas les relations temporelles entre les tâches.

Soit $extTaskSiblingTask$ cette seconde fonction d'extension.
$extTaskSiblingTask : TASKTREE \rightarrow \mathcal{P}(TASKTREE)$
$\forall task \in TASKTREE,$
$extTaskSiblingTask(task) = subtasks(parentOf(task))$

Pour conserver les liens avec les autres points de vue, il faut appliquer *completionTask* sur chaque tâche de la nouvelle sélection.

### 4.4.2 Extension suivant la tâche parente : avec complétion

Nous définissons une fonction d'extension suivant la tâche parente intégrant automatiquement la complétion. Soit *extTaskParent* et *extTaskSibling* les deux fonctions intégrants la complétion et faisant écho aux fonctions définies précédemment. Elles sont définies comme suit :

$extTaskParent : TASKTREE \times SELECTION \rightarrow SELECTION$
$\forall task \in TASKTREE, \forall sel = (\{uie_{chosen}\}, \{appElem_{chosen}\}, \{task_{chosen}\}) \in SELECTION, extTaskParent(task, sel) =$
$\begin{cases} parentOf(task) = \varnothing \Rightarrow sel \\ \\ parentOf(task) \neq \varnothing \Rightarrow newSel \text{ où} \\ sel_{reduced} \leftarrow sel \setminus (\emptyset, \emptyset, \{task\}) \text{ et } newSel \leftarrow sel_{reduced} \cup completionTask(\{parentOf(task)\}) \end{cases}$

$extTaskSibling : TASKTREE \times SELECTION \rightarrow SELECTION$
$\forall task \in TASKTREE, \forall sel = (\{uie_{chosen}\}, \{appElem_{chosen}\}, \{task_{chosen}\}) \in SELECTION,$
$extTaskSibling(task, sel) = sel \bigcup completionTask(extTaskSiblingTask(task))$

La fonction *extTaskParent* définit un premier niveau d'exploration de la description de l'arbre de tâches de l'application.
Nous l'appliquons ainsi de manière récursive pour augmenter le niveau d'exploration de l'arbre de tâches de l'application. Pour cela, il suffit d'appliquer cette fonction à la tâche parente de la tâche sélectionnée et à la nouvelle sélection pour un second niveau d'exploration (cf. figure 4.10, puis au "grand-parent" de la tâche sélectionnée et la nouvelle nouvelle sélection pour un troisième niveau d'exploration etc. Nous définissons les niveaux d'exploration en ajoutant un indice d'exploration :

$extTaskParent : TASKTREE \times SELECTION \times INTEGER \rightarrow SELECTION$
$\forall task \in TASKTREE, \forall sel \in SELECTION, \forall deep \in INTEGER \wedge deep \geqslant 0,$
$extTaskParent(task, sel, 0) = sel$ et
$extTaskParent(task, sel, deep) = extTaskParent(parentOf^{deep-1}(task), extTaskParent(task, sel, deep - 1))$

- Premier niveau d'exploration : $extTaskParent(task, sel, 1) = extTaskParent(task, sel)$
- Second niveau d'exploration :
  $extTaskParent(task, sel, 2) = extTaskParent(parentOf(task), extTaskParent(task, sel, 1))$
- Troisième niveau d'exploration : $extTaskParent(task, sel, 3)$
  $= extTaskParent(parentOf^2(task), extTaskParent(task, sel, 2))$
  $= extTaskParent(parentOf(parentOf(task)), extTaskParent(parentOf(task),$
  $parentOf(parentOf(task), extTaskParent(task, sel, 1)))$

– etc.

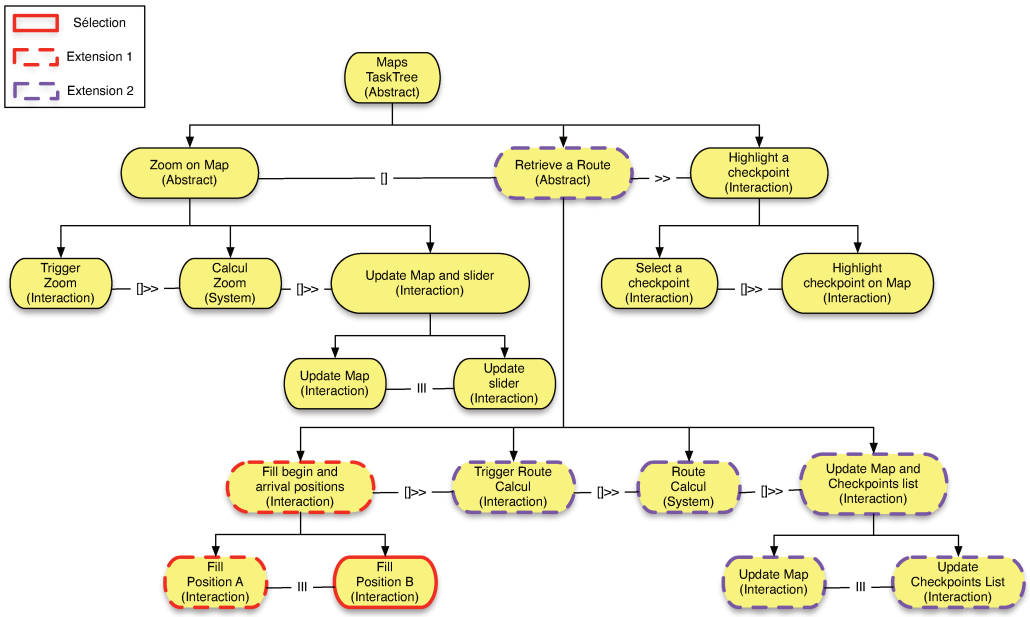



FIGURE 4.10 – Illustration de "l'extension suivant la tâche parente" de niveau 2 appliquée sur l'application "Maps".

### 4.4.3 Extension suivant la tâche parente : à partir d'un élément graphique

Ayant défini la sélection "suivant la tâche parente", nous étudions le cas où nous appliquons cette extension "suivant la tâche parente" à partir d'une sélection d'éléments graphiques.

Dans un premier temps, nous appelons $completionUI$ afin de récupérer les tâches liées aux éléments graphiques sélectionnés. Puis nous appliquons l'extension "suivant la tâche parente" avec complétion sur toutes les tâches de la sélection afin d'une part, de récupérer les nouvelles tâches issues de l'extension, et d'autre part, de remonter à l'interface graphique de l'application en récoltant les éléments graphiques liés aux nouvelles tâches.

Soit $extTaskParentFromUI : APPUI \rightarrow SELECTION$ la fonction d'extension. Elle est définie par l'algorithme 4.

**Algorithme 4** $extTaskParentFromUI(\{uie_{chosen}\})$

**ENTRÉES:** $\{uie_{chosen}\} \in \mathcal{P}(APPUI)$.
**SORTIES:** $sel = (\{uie_{completed}\}, \{appElem_{completed}\}, \{task_{completed}\}) \in SELECTION$

1: $sel \leftarrow completionUI(\{uie_{chosen}\})$
2: $sel \leftarrow \bigcup_{newTask \in \{task_{completed}\}} \{extTaskParent(newTask, sel)\}$
3: $sel \leftarrow sel \bigcup completionTask(\{task_{completed}\})$
4: **retourner** $sel$

Un exemple de cette extension est illustré sur la figure 4.11 (page 81), à partir de la sélection de l'élément graphique "RouteCalculButton", la fonction est appliquée et permet de récupérer dans la sélection sept tâches et quatre éléments graphiques supplémentaires. Pour ne pas surcharger cette figure, les éléments logiciels ne sont pas représentés, mais ils sont bien présents dans les sélections.

### 4.4.4 Extension suivant les relations temporelles : principe

Pour effectuer ce nouveau type d'extension de sélection, nous nous basons sur les relations temporelles des tâches de l'application. A partir d'une tâche et d'un "opérateur temporel" donné, cette extension permet d'ajouter à la sélection toutes les tâches liées par "l'opérateur temporel" donné par rapport à la tâche sélectionnée (cf. figure 4.12).

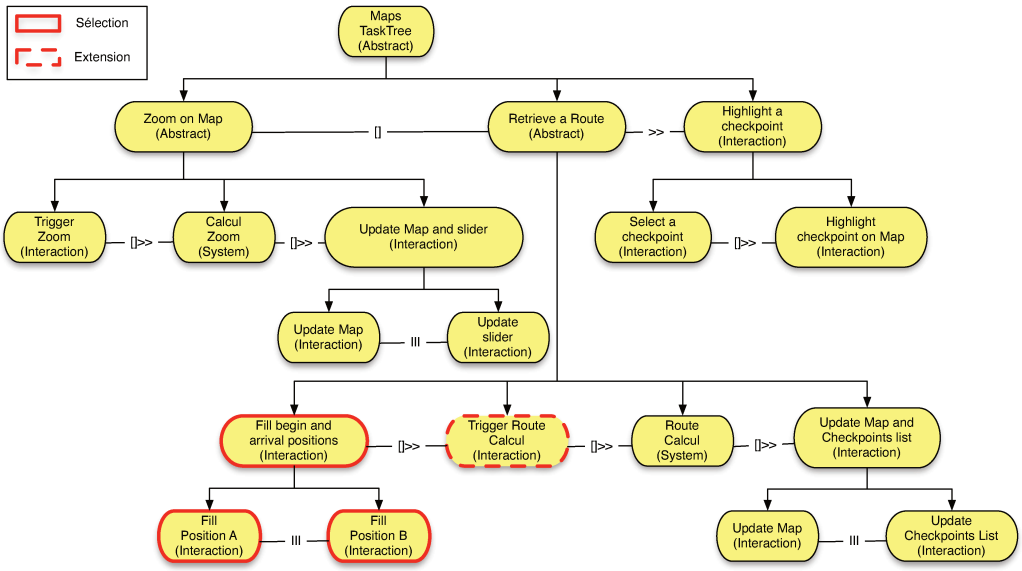



FIGURE 4.12 – Illustration de "l'extension suivant les relations temporelles des tâches" appliquée sur l'application "Maps".

Soit $extTempOpTask$ cette fonction d'extension suivant une relation temporelle précise.
$extTempOpTask : TASKTREE \times TEMPOP \rightarrow \mathcal{P}(TASKTREE)$
$\forall task \in TASKTREE, \forall top \in TEMPOP, extTempOpTask(task,top) = \{task\} \bigcup \{t_j \in TASKTREE, top(task,t_j) \vee (\exists t \in TASKTREE isTaskAncestorOf(t,t_j) \wedge top(task,t))\}$

Nous appliquons $completionTask$ sur les tâches de la nouvelle sélection.

### 4.4.5 Extension suivant les relations temporelles : avec complétion

Nous définissons une fonction d'extension suivant les relations temporelles intégrant automatiquement la complétion. Soit $extTempOp$ cette fonction :

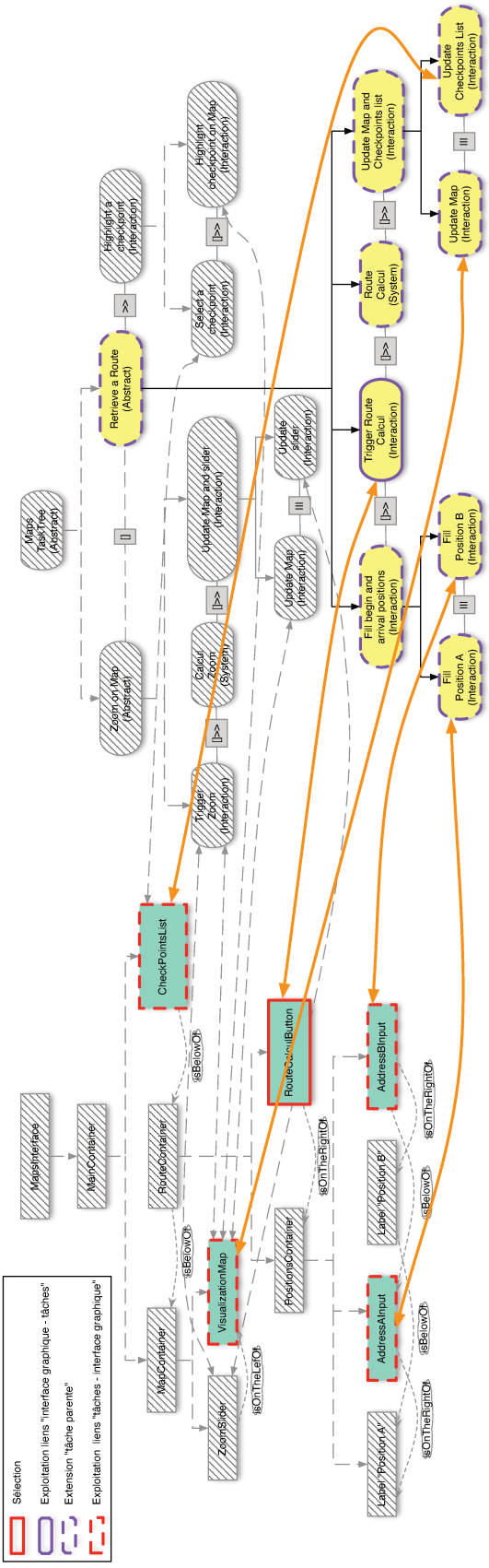



FIGURE 4.11 – Illustration de "l'extension suivant la tâche parente" à partir de la sélection d'un élément graphique, appliquée sur l'application "Maps" (les éléments logiciels sont masqués).

$$extTempOp : TASKTREE \times TEMPOP \times SELECTION \rightarrow SELECTION$$
$\forall task \in TASKTREE, \forall top \in TEMPOP, \forall sel = (\{uie_{chosen}\}, \{appElem_{chosen}\}, \{task_{chosen}\}) \in SELECTION, task \in \{task_{chosen}\}, extTempOp(task, top, sel) =$

$$\begin{cases} extTempOpTask(task) = \varnothing \Rightarrow sel \\ \\ extTempOpTask(task) \neq \varnothing \Rightarrow newSel \text{ où} \\ sel_{completed} \leftarrow completionTask(extTempOpTask(task)) \\ \text{ et } newSel \leftarrow sel_{completed} \cup sel \end{cases}$$

Cette fonction peut s'appliquer récursivement sur chacune des tâches trouvées. Ainsi, nous explorons, comme dans le cas de l'extension suivant la tâche parente, la description de l'arbre de tâches de l'application, non plus de manière hiérarchique, mais selon les relations temporelles. Nous définissons les niveaux suivants en ajoutant la profondeur d'exploration à cette fonction :

$$extTempOp : TASKTREE \times TEMPOP \times SELECTION \times INTEGER \rightarrow SELECTION$$
$\forall task \in TASKTREE, \forall top \in TEMPOP, \forall sel \in SELECTION,$ task étant dans sel, $\forall deep \in INTEGER \wedge deep \geqslant 0,$
$extTempOp(task, top, sel, 0) = sel$ et
$extTempOp(task, top, sel, deep) = \bigcup_{newTask \in \{task_{new}\}} \{extTempOp(newTask, top, newSel)\}$ où
$newSel \leftarrow (\{uie_{new}\}, \{appElem_{new}\}, \{task_{new}\}) \in extTempOp(task, top, sel, deep - 1)$

Une illustration de cette fonction est présente sur la figure 4.13 à un niveau 3 d'exploration avec comme paramètre la relation temporelle *isInSequenceWith*.

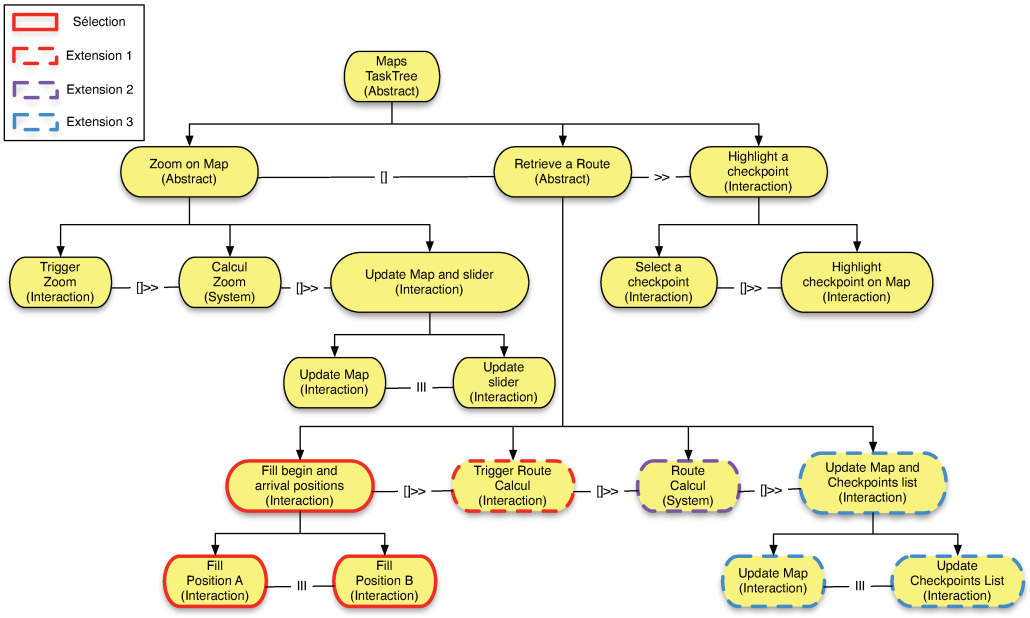



FIGURE 4.13 – Illustration de "l'extension suivant les relations temporelles des tâches" à un niveau 3 d'exploration appliquée sur l'application "Maps".

### 4.4.6 Extension suivant les relations temporelles : à partir d'un élément graphique

Ayant défini la sélection "suivant les relations temporelles", nous étudions le cas où nous appliquons cette extension "suivant les relations temporelles", mais en partant d'une sélection d'éléments graphiques.

Dans un premier temps, nous appelons $completionUI$ afin de récupérer les tâches liées aux éléments graphiques sélectionnés. Puis nous appliquons l'extension "suivant les relations temporelles" avec complétion sur toutes les tâches de la sélection afin d'une part, de récupérer les nouvelles tâches issues de l'extension, et d'autre part, de remonter à l'interface graphique de l'application en récoltant les éléments graphiques liés aux nouvelles tâches. Cette fonction $extTempOpFromUI : APPUI \rightarrow SELECTION$ est définie par l'algorithme 5.

**Algorithme 5** $extTempOpFromUI(\{uie_{chosen}\})$

**ENTRÉES:** $\{uie_{chosen}\} \in \mathcal{P}(APPUI)$.
**SORTIES:** $sel = (\{uie_{completed}\}, \{appElem_{completed}\}, \{task_{completed}\}) \in SELECTION$

1: $sel \leftarrow completionUI(\{uie_{chosen}\})$
2: $sel \leftarrow \bigcup_{newTask \in \{task_{completed}\}} \{extTempOp(newTask, sel)\}$
3: $sel \leftarrow sel \bigcup completionTask(\{task_{completed}\})$
4: **retourner** $sel$

Un exemple de cette extension est illustré par la figure 4.14. A partir de la sélection de l'élément graphique "RouteCalculButton". La fonction est appliquée et permet de récupérer dans la sélection une tâche système "Route Calcul" et aucun autre élément graphique supplémentaire. Appliquée une seconde fois, la sélection est enrichie par une autre tâche "Update Map and Checkpoints list" et deux éléments graphiques "VisualizationMap" et "CheckPointsList". Pour rendre cette sélection opérationnelle il faut appliquer la fonction de consolidation, les éléments graphiques "AddressAInput" et "AddressBInput" seraient alors ajoutés. Les éléments logiciels ne sont pas représentés pour ne pas surcharger la figure, mais ils sont bien présents dans les sélections.

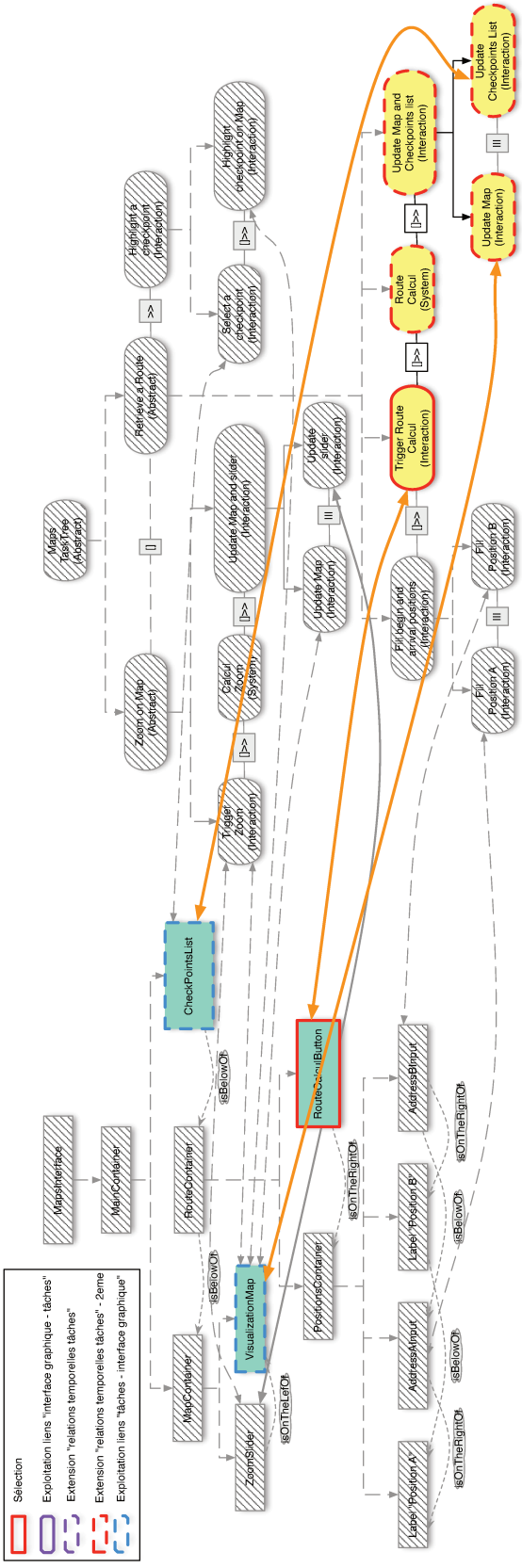



FIGURE 4.14 – Illustration de "l'extension suivant les relations temporelles des tâches" à partir de la sélection d'un élément graphique appliquée sur l'application "Maps" (les éléments logiciels sont masqués).

## 4.5 Combinaison d'extensions

Dans les sections précédentes, nous avons défini un certain nombre de fonctions d'extensions. Celles-ci forment une "boite à outils" que le meneur de la composition peut utiliser pour effectuer la composition d'applications. La figure 4.16 (page 87) présente un exemple de combinaison de deux extensions permettant à partir de la sélection d'un élément graphique d'extraire une sous-partie de l'application "Maps". Dans la figure 4.15 (page 86), nous appliquons $extAppElemFromUI$ à l'élément "RouteCalculBouton". Dans la figure 4.16 (page 87), nous appliquons $extTaskParent$ aux deux tâches sélectionnées. Enfin, sur cette même figure, nous appliquons la fonction de consolidation $consolidate$ afin d'obtenir une sélection opérationnelle.

Le résultat de cette combinaison d'extensions se rapproche du résultat après l'appel de la fonction $completionAppElem$ après l'application de l'extension $extAppElemFromUI$. La différence réside dans les tâches "Fill begin and arrival positions" et "Update Map and Checkpoints list". Celles-ci sont présentes dans la sélection après l'utilisation de la combinaison "$extAppElemFromUI + extTaskParent + consolidate$", mais elles ne sont pas dans la sélection après l'utilisation de "$extAppElemFromUI + completionAppElem$".

## 4.6 Conclusion

Nous avons défini dans ce chapitre les algorithmes de la complétion et la consolidation des sélections. Nous exploitons ainsi les liens pour passer d'une sélection d'éléments d'un point de vue à une sélection d'éléments des trois points de vue.
Nous avons également défini plusieurs explorations exploitant les différents points de vue : relation hiérarchique (tâche ou graphique), relation géométrique (graphique), relation temporelle (tâche) et relation fonctionnelle pour les éléments logiciels. Ces explorations deviennent des extensions combinables qui doivent être compléter et consolider pour obtenir une sous-partie opérationnelle.
Nous avons implémenté ces algorithmes dans le prototype OntoCompo (présenté dans le chapitre 6) utilisé pour des expérimentations décrites dans le chapitre 7.

Ces sélections fournissent, à partir d'applications initiales, des sous-parties d'éléments à composer entre-eux. C'est ce que nous présentons dans le chapitre suivant.

## Publications

Les extensions de sélection ont été plus particulièrement présentées dans les 2 publications suivantes :

[BPDFZ+11] Christian Brel, Anne-Marie Pinna-Déry, Catherine Faron-Zucker, Philippe Renevier, and Michel Riveill. OntoCompo : An Ontology-Based Interactive System To Compose Applications. In *Seventh International Conference on Web Information Systems and Technologies (WEBIST 2011), short paper*, pages 322–327. Springer-Verlag, May 2011.

[BRO+10] Christian Brel, Philippe Renevier, Audrey Occello, Anne-Marie Pinna-Déry, Catherine Faron-Zucker, and Michel Riveill. Application Composition Driven By UI Composition. In *3rd International Conference on Human Computer Software Engineering (HCSE 2010)*, volume 6409 of *LNCS*, pages 198–205. LNCS, October 2010.

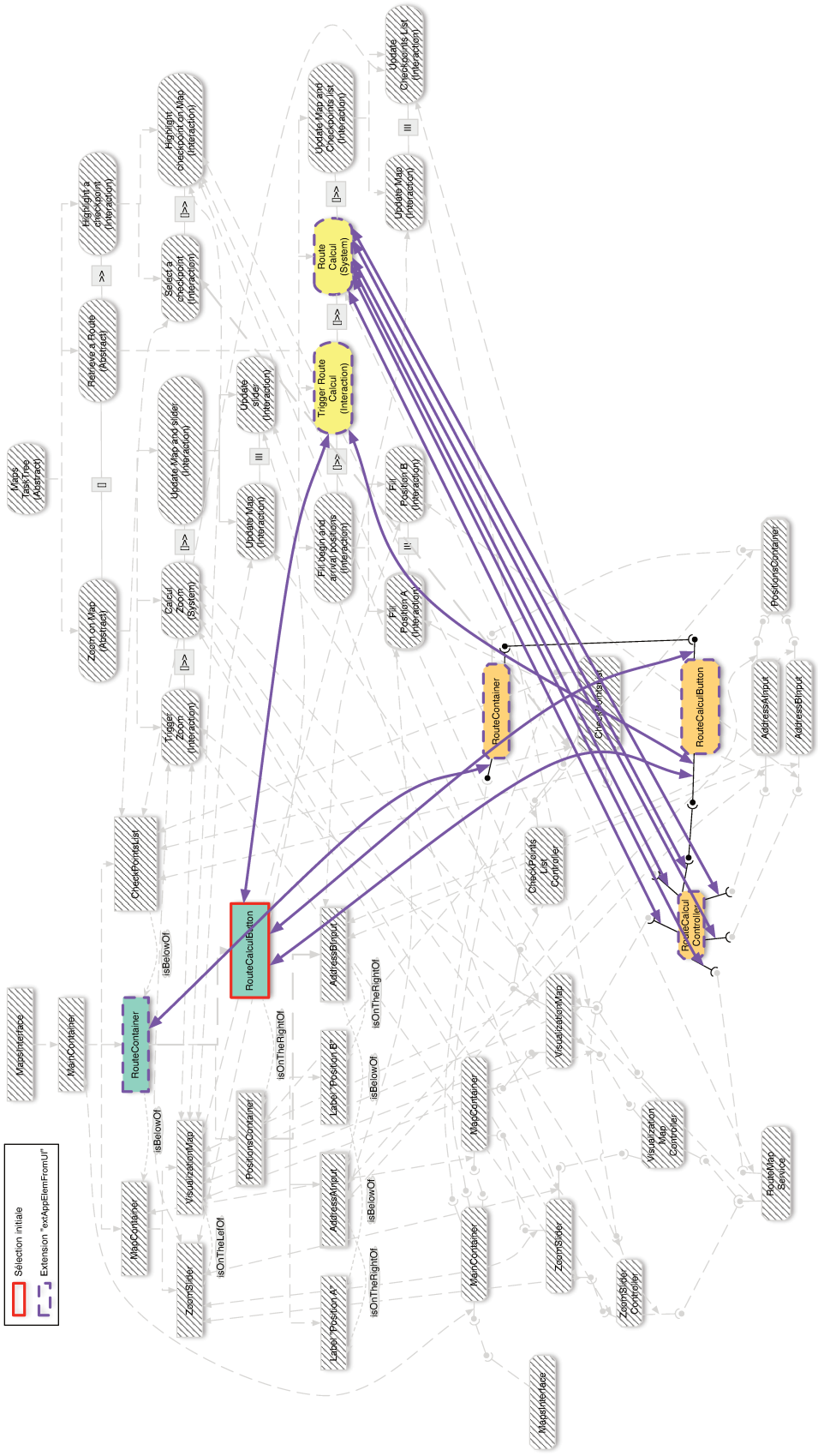



FIGURE 4.15 – Illustration de l'extension *extAppElemFromUI* à partir de la sélection de l'élément graphique "RouteCalculBouton" de l'application "Maps".

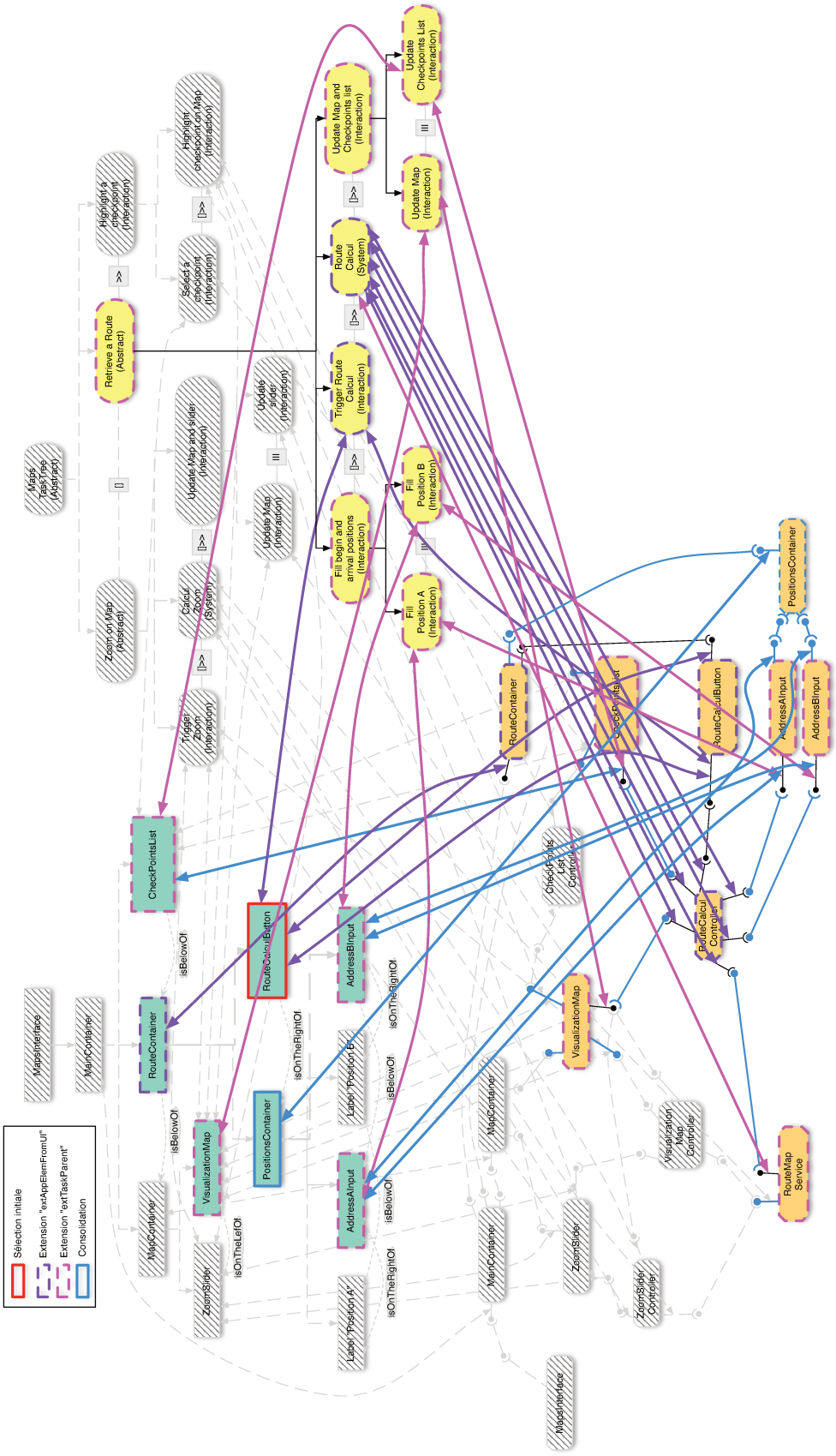



FIGURE 4.16 – Illustration de la combinaison des extensions *extAppElemFromUI* et *extTaskParent* à partir de la sélection de l'élément graphique "RouteCalculBouton" de l'application "Maps".

CHAPITRE 5

# Composition par substitution de parties d'application

**Sommaire**



NOUS définissons dans ce chapitre la substitution qui permet de remplacer un élément sélectionné (graphique, logiciel ou tâche) par un autre élément "équivalent". Cette substitution est réalisée par l'ajout d'une nouvelle entité adaptant l'élément conservé à l'utilisation de l'élément remplacé. Ainsi nous mettons en place les liens entre les applications initiales. La composition sera opérationnelle lorsque les liens entre les éléments logiciels conservés seront établis. Ainsi toute substitution doit être traduite en une substitution entre éléments logiciels.

Lorsque un élément graphique est substitué par un autre, chaque élément logiciel associé à chaque élément graphique impliqué dans cette substitution est récupéré grâce aux liens que nous avons mis en place entre nos modèles. La substitution entre les deux éléments logiciels ainsi récupérés est alors effectuée.

De même, lorsque une substitution entre deux tâches est réalisée, les éléments logiciels associés à chacune des tâches sont récupérés. La différence avec la substitution entre deux éléments graphiques réside dans le fait qu'à cette étape il a potentiellement plusieurs éléments logiciels liés à une tâche. La substitution entre deux tâches revient alors à effectuer des substitutions successives de chacun des éléments logiciels liés à la tâche à supprimer par un des éléments logiciels liés à la tâche à garder. Cette suite de substitutions peut alors être définie par un algorithme externe ou directement par le meneur de la composition etc.

Par conséquent, nous étudions dans les sections suivantes la substitution uniquement entre deux éléments logiciels en commençant par une substitution de granularité plus fine, celle entre deux ports d'éléments logiciels. Puis nous élargissons la substitution au remplacement d'un élément logiciel par un ensemble d'éléments logiciels. Avant de conclure ce chapitre, nous illustrons des substitutions dans notre cadre d'étude.

## 5.1 Substitutions entre deux ports d'éléments logiciels

Cette section définie la substitution entre deux ports d'éléments logiciels. Nous limitons cette substitution aux ports fournis. En effet, dans notre cadre d'étude exposé en introduction de cette thèse (cf.

chapitre 1), nous avons décidé ne pas avoir accès au code source des applications. Or, pour remplacer un requis d'un élément logiciel à un autre, cela signifie modifier le comportement du second pour qu'il puisse exploiter le nouveau requis. Pour faire cela, il faut à priori modifier le code source.

Nous définissons donc les conditions pour que la substitution entre deux ports soit possible, un port devant être **substitué** par l'autre (le port **gardé**).

La première condition est que les deux ports doivent être tous deux fournis.

Dans le cadre de la composition dirigée par les interfaces graphiques, la seconde condition est sur la compatibilité des rôles des ports. Si le rôle du port *substitué* est "UIComponent" (cf. section 3.1.4) alors le rôle du port *gardé* doit aussi être "UIComponent". Le port *substitué* fournissant un élément graphique à insérer dans l'interface graphique, il est nécessaire que le port *gardé* fournisse également un élément graphique. Dans ce cas, la troisième condition n'est pas nécessaire.

La troisième condition est sur la compatibilité des types des ports. Cette compatibilité se valide à partir du type du port *substitué*, $OUTPUT$, $INPUT$ ou $TRIGGER$. Nous décrivons ci-dessous cette compatibilité pour chaque type de port et la substitution proposée pour chacun des cas. Chaque substitution ajoute un élément logiciel que nous appelons "adapteur". Celui-ci joue un rôle différent selon la substitution.

### Le port substitué est de type $OUTPUT$

Si le port *substitué* est de type $OUTPUT$ (cf. figure 5.1) c'est-à-dire proposant le moyen d'accepter des données (pour qu'elles soient affichées ou enregistrées par exemple) alors seul un port de type $OUTPUT$ peuvent substituer celui-ci. Ceci est illustré par la figure 5.1. Dans ce cas, l'adapteur peut avoir plusieurs comportements comme servir au port gardé *gardé* la dernière donnée reçue ou concaténer les données reçues puis servir le résultat au port *gardé* etc.

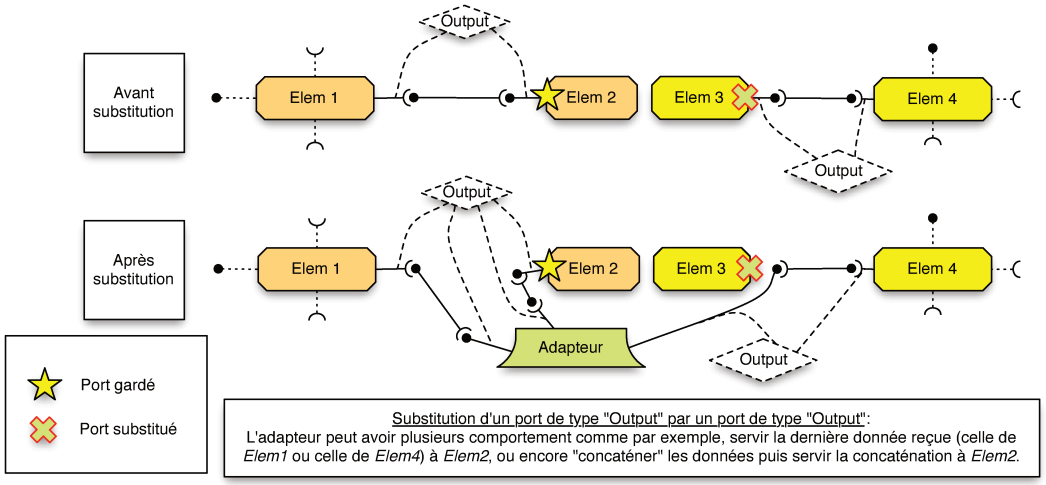



FIGURE 5.1 – Substitution possible entre deux ports, le port *substitué* est de type $OUTPUT$.

### Le port substitué est de type *INPUT*

Si le port *substitué* est de type *INPUT* (cf. figure 5.2) c'est-à-dire proposant le moyen de récupérer des données (fournies par exemple par l'utilisateur à travers l'interface graphique, ou encore fournies par un service), alors seuls les ports de type *INPUT* et *OUTPUT* peuvent substituer celui-ci, comme l'illustre la figure 5.2 :

- Si le port *gardé* est de type *INPUT*, lors de la demande de récupération de la donnée sur l'adapteur, celui-ci retourne la donnée telle quelle au port requis *INPUT* connecté au port *gardé*. Il adapter la donnée pour le port requis *INPUT* connecté au port *substitué*.
- Si le port *gardé* est de type *OUTPUT* alors l'adapteur stocke la donnée afin de la servir telle quelle au port *OUTPUT* et va l'adapter pour la fournir aussi au port requis *INPUT* connecté au port *substitué* à sa demande.

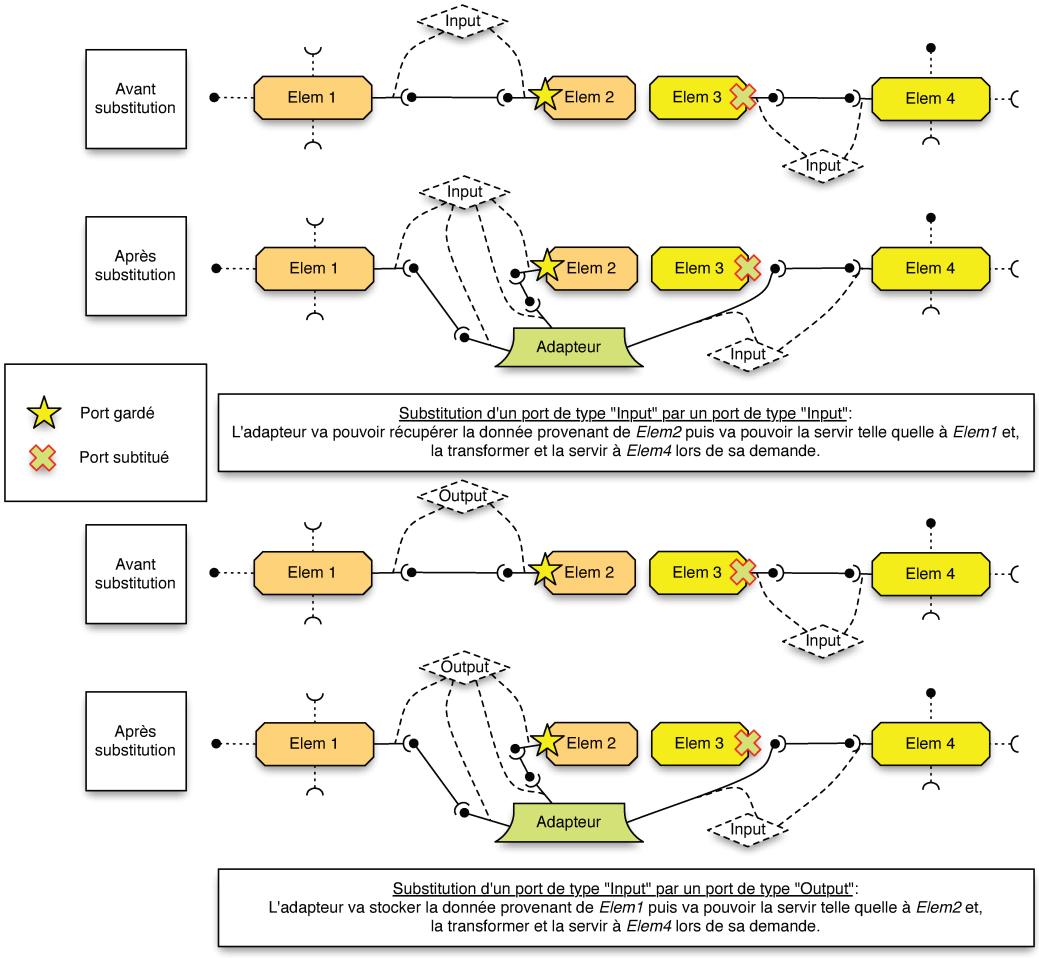



FIGURE 5.2 – Substitutions possibles entre deux ports, le port *substitué* est de type *INPUT*.

### Le port substitué est de type *TRIGGER*

Si le port *substitué* est de type *TRIGGER* (cf. figure 5.3) c'est-à-dire proposant le moyen de déclencher des actions alors quelque soit le type du port *gardé*, la substitution est possible :

- Si le port *gardé* est de type *INPUT*, lorsque la donnée sera récupérée à travers le port *INPUT* de l'adapteur, ce dernier réalise deux actions simultanées. La première est d'effectuer cette demande au port *gardé INPUT* et retourner la réponse en retour au demandeur. La seconde est de déclencher l'action sur le port requis *TRIGGER* connecté au port *substitué*. Le déclenchement se fait donc lors de la solicitation du port de type *INPUT*.
- Si le port *gardé* est de type *OUTPUT*, de manière similaire au cas précédent, lorsque l'adapteur reçoit la donnée, il déclenche, en plus de transférer les données au port *OUTPUT* de l'élément conservé, l'action du port requis *TRIGGER* connecté au port *substitué*.
- Enfin, si le port *gardé* est de type *TRIGGER*, l'adapteur peut avoir plusieurs comportements comme déclencher les actions des deux ports requis *TRIGGER* connectés à celui-ci en parallèle, en séquence, etc.

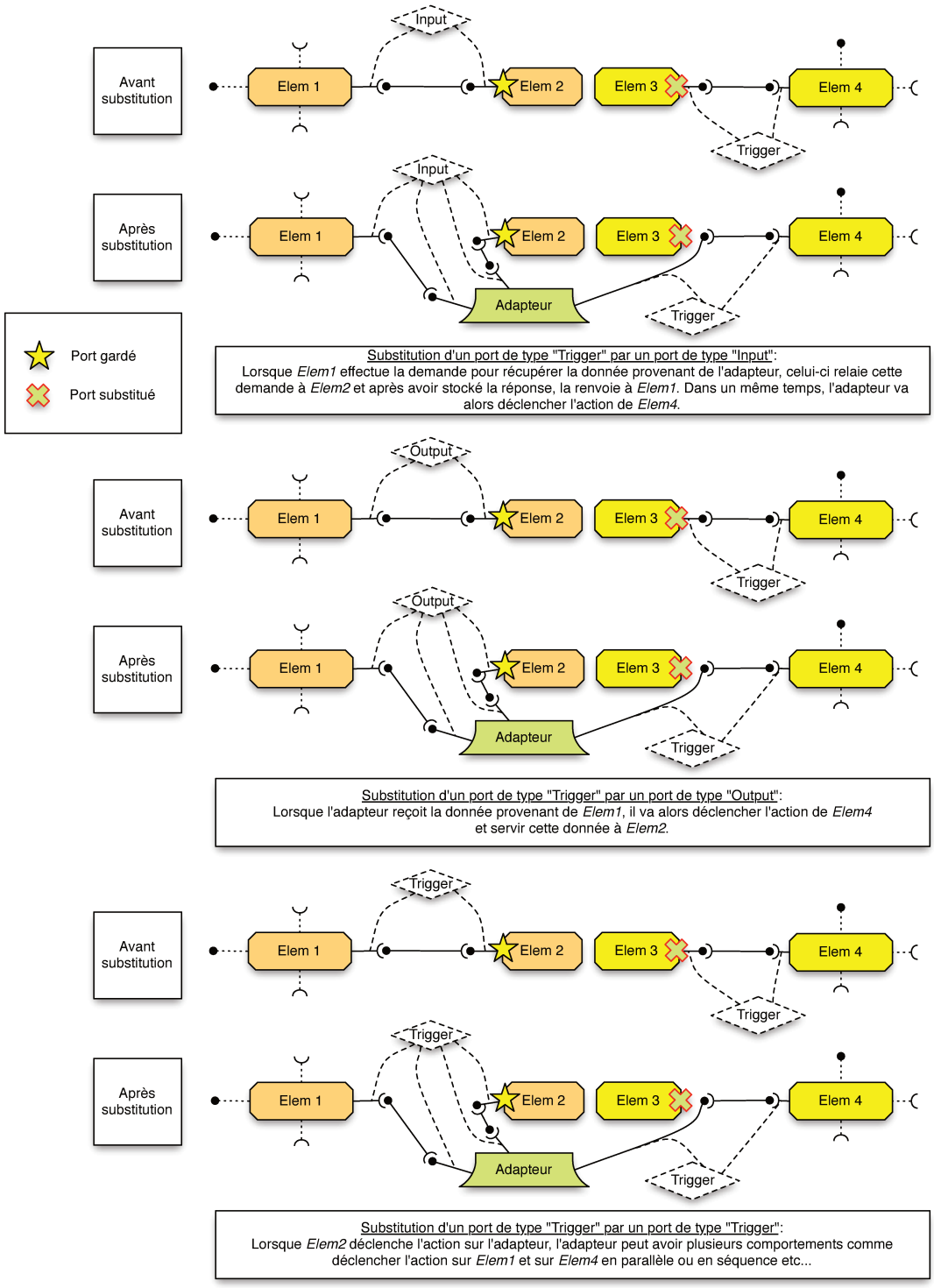



FIGURE 5.3 – Substitutions possibles entre deux ports, le port *substitué* est de type *TRIGGER*.

Afin de pouvoir décrire ces conditions de substitution, nous définissons la fonction *isSubstitutableBy* permettant de savoir si un port peut être substitué par un autre. Dans la suite, $P_{kept}$ représente le port *gardé* et $P_{removed}$, le port *substitué*.

*isSubstitutableBy* : $PORTS \times PORTS \in BOOLEAN, \forall P_{kept} \in PORTS, \forall P_{removed} \in PORTS, isProvided(P_{kept}) = isProvided(P_{removed}) = TRUE \wedge isUIComponentPort(P_{removed}) \Rightarrow isUIComponentPort(P_{kept})$
*isSubstitutableBy* permet de savoir si $P_{removed}$ peut être substitué par $P_{kept}$, dans le détail :
- $isInput(P_{removed}) \wedge (isInput(P_{kept}) \vee isOutput(P_{kept})) \Rightarrow isSubstitutableBy(P_{kept}, P_{removed}) \leftarrow TRUE$
- $isOutput(P_{removed}) \wedge isOutput(P_{kept}) \Rightarrow isSubstitutableBy(P_{kept}, P_{removed}) \leftarrow TRUE$
- $isTrigger(P_{removed}) \Rightarrow isSubstitutableBy(P_{kept}, P_{removed}) \leftarrow TRUE$

### Définitions de fonctions de manipulation de port

Avant de définir la fonction de substitution entre deux ports, nous définissons quelques fonctions simplifiant sa définition. La première d'entre-elles est une fonction de duplication de port $duplicatePort$ : $PORTS \rightarrow PORTS$ définie par l'algorithme 6.

**Algorithme 6** $duplicatePort(port)$

**ENTRÉES:** $port \in PORTS$.
**SORTIES:** $dport \in PORTS$ tel que $dport$ est le duplicata de $port$, sans connexion.

1: $dport \in PORTS$
2: $isRequired(dport) \leftarrow isRequired(port)$
3: $isProvided(dport) \leftarrow isProvided(port)$
4: $typePort(dport) \leftarrow typePort(port)$
5: $rolePort(dport) \leftarrow rolePort(port)$
6: $connexions(dport) \leftarrow \varnothing$
7: **retourner** $dport$

Nous définissons une fonction permettant de connecter 2 ports :

*connect* : $PORTS \times PORTS \rightarrow BOOLEAN$, cette fonction permet de connecter deux ports : $\forall p, q \in PORTS, connect(p,q) \Leftrightarrow p$ et $q$ sont connectés, tel que : $areConnected(p,q) == TRUE$

Nous définissons une fonction permettant de déconnecter 2 ports :

*disconnect* : $PORTS \times PORTS \rightarrow BOOLEAN$, cette fonction permet de déconnecter deux ports :
$\forall p, q \in PORTS, disconnect(p,q) \Leftrightarrow p$ et $q$ sont déconnectés, tel que : $areConnected(p,q) == FALSE$

Nous définissons une fonction $createConnectablePort : PORTS \rightarrow PORTS$ de création de port, permettant d'obtenir un port que l'on peut connecter au port donné (cf. algorithme 7).

**Algorithme 7** $createConnectablePort(port)$

**ENTRÉES:** $port \in PORTS$.
**SORTIES:** $cport \in PORTS$ tel que $cport$ est le port que l'on peut connecter à $port$.

1: $cport \in PORTS$
2: $isRequired(cport) \leftarrow \neg isRequired(port)$
3: $isProvided(cport) \leftarrow \neg isProvided(port)$
4: $typePort(cport) \leftarrow typePort(port)$
5: $rolePort(cport) \leftarrow rolePort(port)$
6: $connexions(cport) \leftarrow \varnothing$
7: **retourner** $cport$

Nous définissons également une fonction permettant de récupérer tous les ports actifs (c'est-à-dire connectés à un autre port) d'un élément logiciel.

Soit $activePorts$ cette fonction.
$activePorts : APPLI \rightarrow \mathcal{P}(PORTS)$,
$\forall appelem \in APPLI, activePorts(appelem) = \{port \in PORTS, / connexions(port) \neq \varnothing\}$

Finalement, nous définissons une fonction permettant de faire la liste des ports éligibles d'un éléments logiciel, *i.e.*, pouvant remplacer le port donné en paramètre. Cette fonction $eligiblePorts$ : $APPLI \times PORTS \rightarrow \mathcal{P}(PORTS)$ est définie par l'algorithme 8. [1]

**Algorithme 8** $eligiblePorts(appelem, port)$

**ENTRÉES:** $appelem \in APPLI, port \in PORTS$.
**SORTIES:** $listEligiblePorts \in \mathcal{P}(PORTS)$ tel que $listEligiblePorts$ est l'ensemble des ports pouvant remplacer $port$.

1: $listEligiblePorts \leftarrow \varnothing$
2: **pour tout** $P_{potential} \in portsFromAppElem(appelem)$ **faire**
3: **si** $isSubstitutableBy(P_{potential}, port)$ **alors**
4: $appendList(listEligiblePorts, P_{potential})$
5: **finsi**
6: **fin pour**
7: **retourner** $listEligiblePorts$

## Définition de la fonction de substitution

Nous définissons $substPort$, la fonction de substitution entre 2 ports. Elle produit l'adapteur, qui est un élément logiciel permettant de réaliser la substitution du port *substitué* par le port *gardé*.

$substPort : PORTS \times PORTS \rightarrow APPLI$,
$\forall P_{kept} \in PORTS, \forall P_{removed} \in PORTS$,
$substPort(P_{kept}, P_{removed}) = adapter, adapter \in APPLI / isSubstitutableBy(P_{kept}, P_{removed}) \Rightarrow$
$adapter \leftarrow createAdapter(P_{kept}, P_{removed})$

1. Nous utilisons la fonction $appendList$ qui ajoute un élément à une liste.

*createrAdapter* est une fonction définie par l'algorithme 9.

**Algorithme 9** $createAdapter(P_{kept}, P_{removed})$

**ENTRÉES:** $P_{kept}, P_{removed} \in PORTS$.
**SORTIES:** $adapter \in APPLI$ tel que $adapter$ est l'adapteur entre le port $P_{kept}$ et $P_{removed}$.

1: $adapter \in APPLI$
2: $duplicatePortP_{removed} \leftarrow duplicatePort(P_{removed})$
3: $associateAppElemPort(adapter, duplicatePortP_{removed})$
4: **pour tout** $port \in connexions(P_{removed})$ **faire**
5: $disconnect(P_{removed}, port)$
6: $connect(duplicatePortP_{removed}, port)$
7: **fin pour**
8: $connectablePortP_{kept} \leftarrow createConnectablePort(P_{kept})$
9: $associateAppElemPort(adapter, connectablePortP_{kept})$
10: $duplicatePortP_{kept} \leftarrow duplicatePort(P_{kept})$
11: $associateAppElemPort(adapter, duplicatePortP_{kept})$
12: **pour tout** $port \in connexions(P_{kept})$ **faire**
13: $disconnect(P_{kept}, port)$
14: $connect(duplicatePortP_{kept}, port)$
15: **fin pour**
16: $connect(connectablePortP_{kept}, P_{kept})$
17: **retourner** $adapter$

Les différentes étapes de cet algorithme sont illustrées sur la figure 5.4. Les étapes 1 à 3 de l'algorithme créent un adapteur ayant un port fourni, copie du port *substitué*. Les étapes 4 à 7 déconnectent tous les ports connectés au port *substitué* et les connectent à l'adapteur (à la copie du port *substitué*). Les étapes 8 à 11 créent deux nouveaux ports à l'adapteur : le premier est une copie du port *gardé* et le second est un port que l'on peut connecter au port *gardé*. Les étapes 12 à 15 déconnectent tous les ports connectés au port *gardé* et les connectent à l'adapteur (à la copie du port *gardé*). Enfin, l'étape 16 connecte le port *gardé* à l'adapteur.

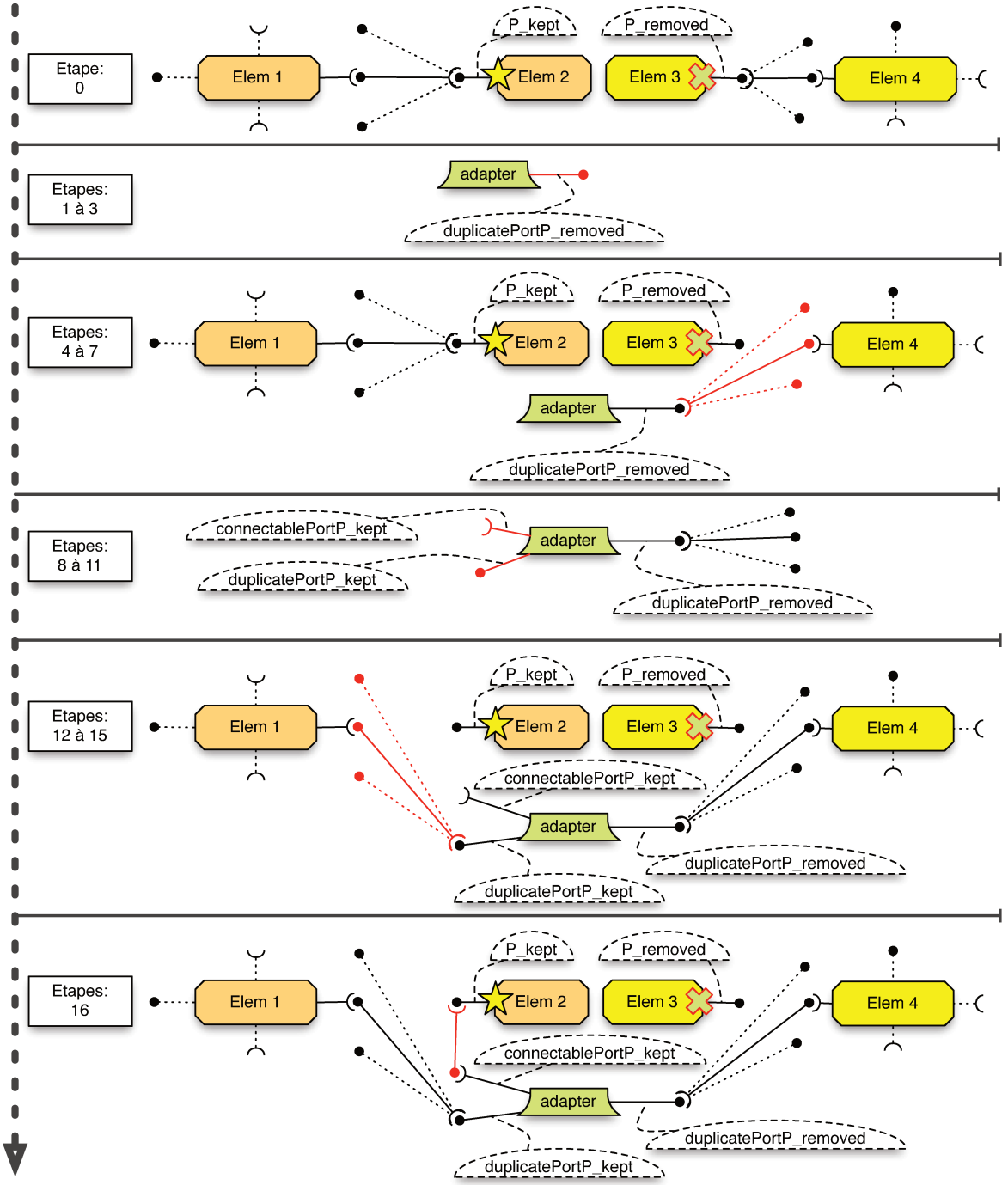



FIGURE 5.4 – Illustration des étapes de l'algorithme de la fonction *createAdapter*.

## 5.2 Substitution entre 2 éléments logiciels

Après avoir traité la substitution entre 2 ports d'éléments logiciels, nous définissons la substitution entre 2 éléments logiciels. Cette substitution remplace un élément logiciel par un autre. Pour cela, nous procédons par substitutions successives des ports fournis de l'élément *substitué* par les ports fournis de l'élément *gardé*.

### Définition de la fonction de substitution

Soit $substElem : APPLI \times APPLI \rightarrow \mathcal{P}(APPLI)$ la fonction de substitution d'un élément logiciel par un autre. Cette fonction est défini par l'algorithme 10. [2]

**Algorithme 10** $substElem(AppElem_{kept}, AppElem_{removed})$

**ENTRÉES:** $AppElem_{kept}, AppElem_{removed} \in APPLI$.
**SORTIES:** $Adapters \in \mathcal{P}(APPLI)$ tel que $Adapters$ est l'ensemble des adapteurs nécessaires pour substituer $AppElem_{removed}$ par $AppElem_{kept}$.

1: $Adapters \leftarrow \varnothing$
2: $activePorts_{removed} \leftarrow activePorts(AppElem_{removed})$
3: **pour tout** $port \in activePorts_{removed}$ **faire**
4: $listEligiblesPorts \leftarrow eligiblePorts(AppElem_{kept}, port)$
5: **si** $sizeOf(listEligiblesPorts) = 0$ **alors**
6: **retourner** *null*
7: **sinon**
8: **si** $sizeOf(listEligiblesPorts) = 1$ **alors**
9: $theOne \leftarrow first(listEligiblesPorts)$
10: **sinon**
11: $theOne \leftarrow chooseEligiblePort(listEligiblesPorts, port)$
12: **finsi**
13: $adapter \leftarrow substPort(theOne, port)$
14: $appendList(Adapters, adapter)$
15: **finsi**
16: **fin pour**
17: **retourner** *Adapters*

Tout d'abord, nous remarquons que cet algorithme termine sans rien modifier dans le cas où un port de l'élément logiciel à *substituer* ne peut pas être substitué.
Ensuite, nous utilisons dans cet algorithme, la fonction *chooseEligiblePort* qui permet de déterminer le port à utiliser parmi la liste des ports éligibles. L'algorithme de cette fonction peut être un algorithme externe ou tout simplement une demande effectuée auprès du meneur de la composition afin qu'il puisse lui-même choisir le port à utiliser. L'algorithme 11 présente un algorithme de choix externe. Cet algorithme très simple recherche dans la liste des ports éligibles, si possible le premier port qui a le même type et le même rôle que le port à *substituer*, sinon le premier port qui a le même type. Si aucun port n'est trouvé, alors il choisit le premier de la liste.

2. Nous utilisons les fonctions : *sizeOf* qui donne la taille d'une liste ; *first* qui renvoie le premier élément d'une liste ; *appendList* qui ajoute un élément à une liste.

**Algorithme 11** $SimpleChoice(listEligiblesPorts, P_{removed})$

**ENTRÉES:** $listEligiblesPorts \in \mathcal{P}(PORTS), P_{removed} \in PORTS$.
**SORTIES:** $theOne \in PORTS$ tel que $theOne$ est le port choisi pour remplacer $P_{removed}$.

1: $portType \leftarrow typePort(port)$
2: $theOne \leftarrow \varnothing$
3: **pour tout** $eligibleP \in listEligiblesPorts$ **faire**
4: $eligiblePType \leftarrow typePort(eligibleP)$
5: **si** $eligiblePType == portType$ **alors**
6: **si** $(isUIPort(P_{removed}) = isUIPort(eligibleP)) \wedge (isUIComponentPort(P_{removed}) = isUIComponentPort(eligibleP))$ **alors**
7: $theOne \leftarrow eligibleP$
8: **retourner** $theOne$
9: **sinon**
10: **si** $theOne = \varnothing$ **alors**
11: $theOne \leftarrow eligiblePort$
12: **finsi**
13: **finsi**
14: **finsi**
15: **fin pour**
16: **si** $theOne = \varnothing$ **alors**
17: $theOne \leftarrow first(listEligiblesPorts)$
18: **finsi**
19: **retourner** $theOne$

## 5.3 Remplacement d'un élément logiciel par un ensemble d'éléments logiciels

Comme nous l'avons vu dans la section précédente, certains cas ne permettent pas de remplacer un élément logiciel par un *seul et unique* autre. Pour palier à cette situation, nous proposons une fonction de substitution qui prend en paramètre l'élément logiciel à *substituer*, et un ensemble d'éléments logiciels pouvant le remplacer. Cet ensemble peut être fourni directement par le meneur de la composition ou par un algorithme externe. La fonction de consolidation (cf. chapitre 4) exploitant les liens entre modèles permet d'établir une liste d'éléments logiciels éligibles pour substituer l'élément logiciel sélectionné.

Nous définissons la fonction de substitution $substMultiElem : \mathcal{P}(APPLI) \times APPLI$ qui permet de substituer un élément logiciel par un ensemble d'éléments logiciels dans l'algorithme 12.

**Algorithme 12** $substMultiElem(ListAppElem_{kept}, AppElem_{removed})$

**ENTRÉES:** $ListAppElem_{kept} \in \mathcal{P}(APPLI), AppElem_{removed} \in APPLI$.
**SORTIES:** $Adapters \in \mathcal{P}(APPLI)$ tel que $Adapters$ est l'ensemble des adapteurs nécessaires pour substituer $AppElem_{removed}$ par $ListAppElem_{kept}$.

1: $Adapters \leftarrow \varnothing$
2: $activePorts_{removed} \leftarrow activePorts(AppElem_{removed})$
3: **pour tout** $port \in activePorts_{removed}$ **faire**
4: **pour tout** $AppElem_{kept} \in ListAppElem_{kept}$ **faire**
5: $adapter \leftarrow testAndSubstElem(AppElem_{kept}, port)$
6: **si** $adapter \neq null$ **alors**
7: $appendList(Adapters, adapter)$
8: BREAK ; Continuer avec le prochain port actif.
9: **sinon**
10: Continuer avec le prochain élément logiciel disponible dans $ListAppElem_{kept}$
11: **finsi**
12: **fin pour**
13: **fin pour**
14: **retourner** $Adapters$

**Définition de** $testAndSubstElem(appelem, port)$

**ENTRÉES:** $appelem \in APPLI, port \in PORTS$.
**SORTIES:** $adapter \in APPLI$ tel que $adapter$ est l'adapteur pour substituer un port de $port$ par un port $appelem$.

1: $listEligiblesPorts \leftarrow eligiblePorts(appelem, port)$
2: **si** $sizeOf(listEligiblesPorts) = 0$ **alors**
3: **retourner** $null$
4: **sinon**
5: **si** $sizeOf(listEligiblesPorts) = 1$ **alors**
6: $theOne \leftarrow first(listEligiblesPorts)$
7: **sinon**
8: $theOne \leftarrow chooseEligiblePort(listEligiblesPorts, port)$
9: **finsi**
10: $adapter \leftarrow substPort(theOne, port)$
11: **retourner** $adapter$
12: **finsi**

## 5.4 Retour sur l'étude de cas

Afin d'illustrer la composition par substitutions, nous détaillons la substitution d'un élément graphique de l'application "Maps" par un élément graphique de l'application "Cinema". Le processus de composition entre ces deux applications est décrit dans l'annexe D.

Nous décrivons la substitution de l'élément graphique "AddressAInput" provenant de l'interface graphique de "Maps" par l'élément graphique "AddressInput" provenant de l'interface graphique de "Cinema".

Pour effectuer cette substitution, nous utilisons les liens entre modèles afin de récupérer les éléments logiciels liés aux éléments graphiques impliqués dans cette substitution. Ainsi comme l'illustre la figure 5.5, nous récupérons l'élément logiciel "AddressAInput" de l'application "Maps" et l'élément logiciel "AddressInput" de l'application "Cinema".

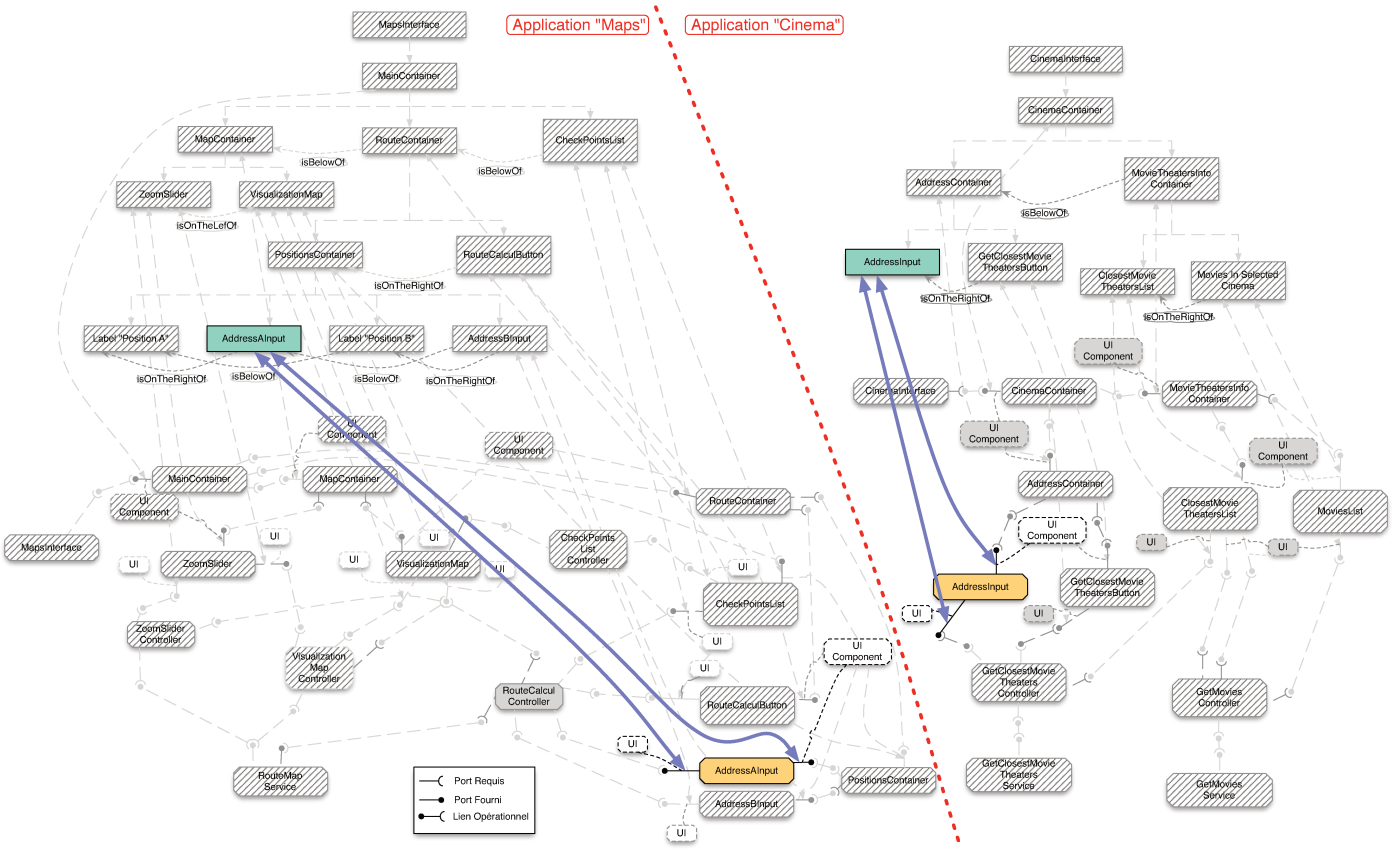



FIGURE 5.5 – Utilisation des liens entre modèles pour récupérer les éléments logiciels liés aux éléments graphiques impliqués dans la substitution.

Cette étape effectuée, la substitution entre deux éléments logiciels utilise l'algorithme 10 (fonction *substElem*) :

– Les ports "actifs" de l'élément logiciel "AddressAInput" sont identifiés.

– Un premier port actif de cet élément est le port de rôle "UI" et de type *INPUT*. Pour le remplacer, l'algorithme recherche les ports éligibles appartenant à l'élément logiciel gardé "AddressInput". Un seul port est éligible, celui ayant pour role "UI" et de type *INPUT*. L'algorithme effectue alors la substitution entre ces deux ports (cf. fonction *substPort* et algorithme 9) comme illustré sur la figure 5.6.
– Un second port actif de l'élément "AddressAInput" est le port de rôle "UIComponent" et de type *INPUT*. Pour le remplacer, l'algorithme recherche les ports éligibles appartenant à l'élément logiciel gardé "AddressInput". Un seul port est éligible, celui ayant pour role "UIComponent" et de type *INPUT*. L'algorithme effectue alors la substitution entre ces deux ports (cf. fonction *substPort* et algorithme 9) comme illustré sur la figure 5.7.

Les deux seuls ports actifs de "AddressAInput" (provenant de l'application "Maps") étant remplacés, cet élément logiciel est désormais totalement déconnecté des autres éléments logiciels (cf. figure 5.7). L'élément logiciel "AddressAInput" et l'élément graphique correspondant sont donc retirés de l'application qui est en train d'être composée.
Un lien est établi entre les deux applications, l'adresse de départ de l'itinéraire de l'application "Maps" étant maintenant donnée par l'adresse renseignée dans l'application "Cinema".

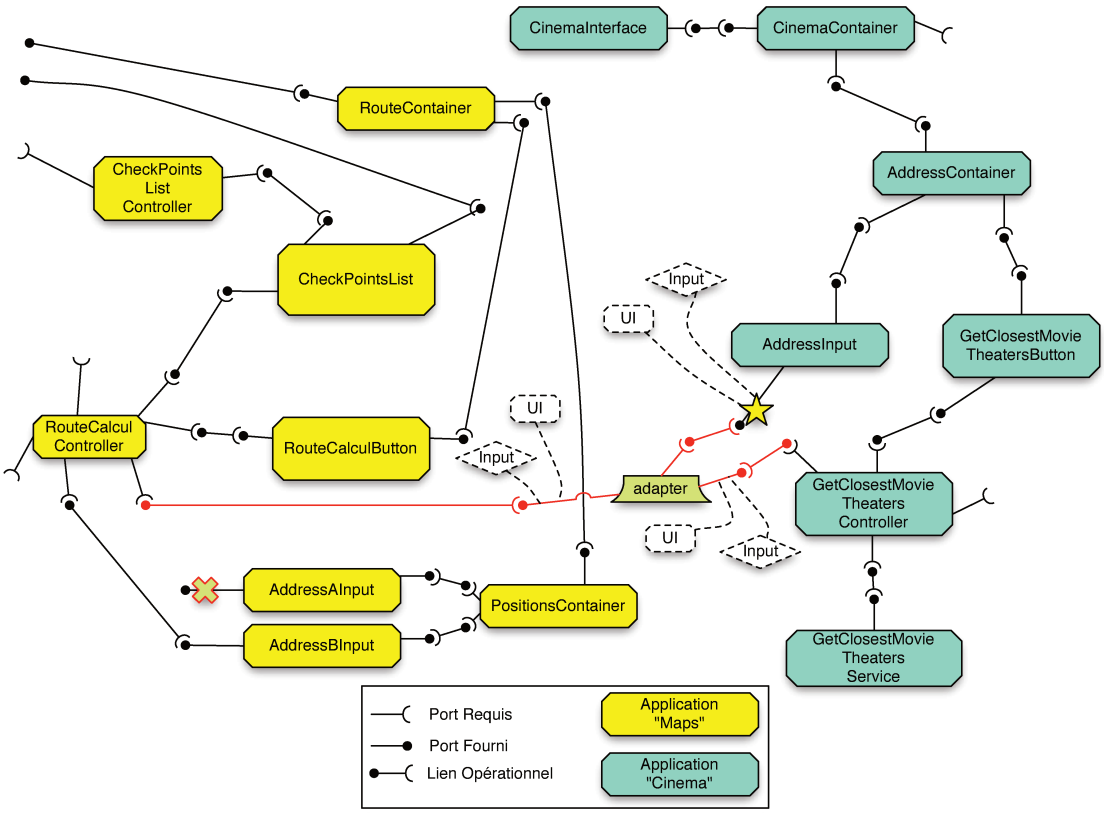



FIGURE 5.6 – Illustration de la création de l'adapteur pour la substitution des ports de type *INPUT* et de rôle "UI".

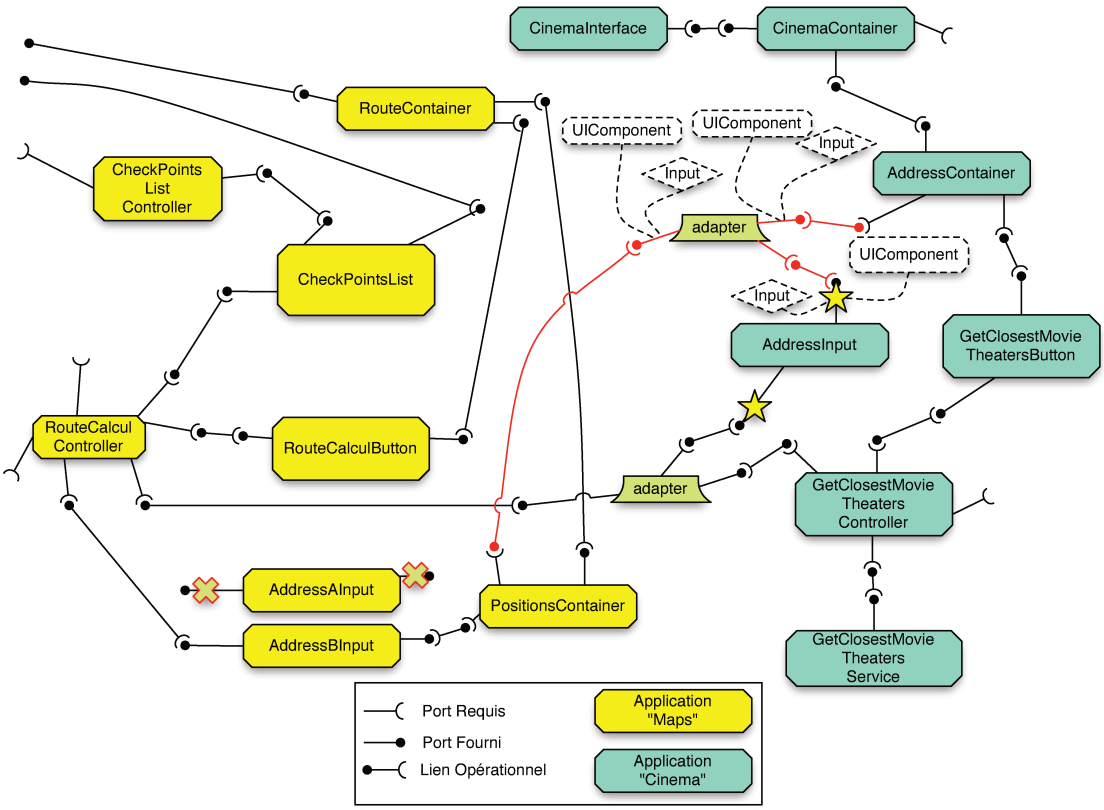



FIGURE 5.7 – Illustration de la création de l'adapteur pour la substitution des ports de type *INPUT* et de rôle "UIComponent".

## 5.5 Conclusion

Dans ce chapitre, nous avons montré que nous pouvions composer les applications en substituant des entités logicielles de l'une par des entités logicielles de l'autre. La substitution se fait au niveau des éléments logiciels qui sont effectivement exécutés. Les substitutions sont guidées par les types des ports concernés et mises en œuvre par insertion d'adapteurs. Ces adapteurs peuvent être complétés une fois la substitution terminée.

La partie suivante décrit le prototype développé dans le cadre de cette thèse puis elle décrit les expérimentations que nous avons conduites.

# Publications

La composition par substitution a été présentée dans les publications suivantes :

### Publications dans des Conférences Internationales

[BRPDR12a] Christian Brel, Philippe Renevier, Anne-Marie Pinna-Déry, and Michel Riveill. Annotated Component-Based Description for Application Composition. In *The Seventh*

*International Conference on Software Engineering Advances (ICSEA 2012)*, November 2012.

[BRPDR12b] Christian Brel, Philippe Renevier, Anne-Marie Pinna-Déry, and Michel Riveill. Application and UI composition using a Component-Based Description and Annotations. In *38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2012)*, pages 204–207, September 2012.

[BRPDR12c] Christian Brel, Philippe Renevier, Anne-Marie Pinna-Déry, and Michel Riveill. UI Modeling as Ontology for Composition. In *The Twenty First International Conference On Software Engineering and Data Engineering (SEDE 2012)*, pages 67–72, June 2012.

### Publications dans des Conférences Nationales

[BM11] Christian Brel and Sébastien Mosser. Vers une approche flot de données pour supporter la composition d'interfaces homme-machine. In *Journées sur l'Ingénierie Dirigée par les Modèles (IDM'11)*, pages 1–7, Lille, June 2011.

# Troisième partie

# Application à travers un prototype et validation

CHAPITRE 6

# OntoCompo : un processus et un prototype pour la composition

**Sommaire**



NOUS décrivons dans ce chapitre la mise en œuvre de notre approche de composition d'applications présentée dans les chapitres précédents. Nous présentons le processus de composition que nous avons choisi pour ce prototype, le développement des applications à composer avec l'utilisation d'ontologies pour leurs descriptions, l'architecture global de l'application de composition mettant en place le processus choisi et enfin, les détails d'implémentation pour chaque étape du processus.

## 6.1 Processus de composition choisi

Les chapitres précédents décrivent en détails notre approche et les différents algorithmes associés. Ces algorithmes peuvent être combinés de différentes manières afin d'effectuer la composition d'applications. Nous avons choisi une combinaison simple de ces algorithmes pour mettre en œuvre notre approche. Le prototype met en œuvre les trois étapes principales : Sélection - Substitution - Placement.

Le prototype propose dans un premier temps de charger les applications à composer. Puis il lui propose de sélectionner les éléments graphiques des applications qu'il veut utiliser pour la composition. A cette étape, les extensions de sélection lui sont proposées afin d'effectuer une sélection opé-

rationnelle. Ensuite, le prototype propose d'effectuer les différentes substitutions. Enfin, le prototype propose une dernière étape de placement des éléments graphiques dans la nouvelle interface.

## 6.2 Implémentation des modèles de description d'une application

Chaque modèle proposé dans le chapitre 3 pour décrire une application a été développé à l'aide d'une ontologie. L'utilisation d'ontologies nous permet de mettre en place rapidement les requêtes nécessaires pour répondre aux différents algorithmes des extensions de sélection. Nous avons défini trois ontologies pour décrire les trois points de vue d'une application : le point de vue "interface graphique", le point de vue "opérationnel", et le point de vue "tâches". Ces ontologies sont représentées à l'aide du standard OWL Lite [1] et sont décrites dans l'annexe B (page 163).

### 6.2.1 Applications en composants Fractal, modèle opérationnel sous forme d'ontologie

Les applications sont développées suivant une architecture à composants, définie par l'implémentation Julia du modèle Fractal. L'ensemble de l'application, que ce soit ses fonctionnalités ou son interface graphique, est implémenté sous forme de composants. Certains composants sont donc des composants constituants l'interface graphique de l'application et sont identifiables par leurs ports particuliers (ayant pour rôle "UI Component" au sein des modèles) manipulants dans le prototype des composants SWING.

*OntoCompo* est l'ontologie pour décrire le modèle opérationnel d'une application. Cette ontologie est constituée de sept classes et dix propriétés. Nous en faisons la description détaillée dans l'annexe B section B.3 (page168).

#### MapsApp.rdf

Ci-dessous est présenté un extrait de la description sémantique du modèle opérationnel de l'application "Maps" - "MapsApp.rdf". Cet extrait est la description de l'élément logiciel "RouteCalculButton". Cet élément est lié à l'élément graphique de même nom. Il a un identifiant qui correspond dans le prototype à un composant fractal constitutif de l'application. De même, chacun de ses ports a un nom qui correspond au nom donné à un port du composant fractal. Ce sont grâce à ces identifiants et aux noms de ports que certains algorithmes décrits dans ce manuscrit sont implémentés. C'est le cas de la génération de l'adapteur lors de la substitution entre deux éléments qui nécessite la génération de code Java. Cette génération produit un composant fractal dont les ports sont déterminés en fonction des composants fractals existants.

Une partie de la description sémantique du modèle opérationnel de l'application "Maps" est présente dans l'annexe C section C.3 (page 176).

Listing 6.1 – **Extrait de la description sémantique au format RDF : fichier MapsApp.rdf.part**

```
115 <AppElement rdf:ID="RouteCalculButton">
116 <!-- Description des ports de l élément logiciel correspondant au bouton -->
117     <hasPort rdf:resource="#GetRouteCalculButton" />
118     <hasPort rdf:resource="#ButtonActionPerformed" />
119
```

1. http ://www.w3.org/TR/owl-features/

```
120     <hasID>maps.graphics.ButtonComposant</hasID>
121 </AppElement>
122
123     <ProvidedPort rdf:ID="GetRouteCalculButton">
124         <hasType rdf:resource="&ontocompo;-instances#Input" />
125         <hasRole rdf:resource="&ontocompo;-instances#UIComponentRole" />
126         <connectedTo rdf:resource="#RouteContainerGetRouteCalculButton" />
127
128         <hasName>b</hasName>
129     </ProvidedPort>
130
131     <ProvidedPort rdf:ID="ButtonActionPerformed">
132         <hasType rdf:resource="&ontocompo;-instances#Trigger" />
133         <hasRole rdf:resource="&ontocompo;-instances#UIRole" />
134         <connectedTo rdf:resource="#RouteCalculControllerSearchRoute" />
135
136         <hasName>trigger</hasName>
137     </ProvidedPort>
```

### 6.2.2 Interface graphique en Java/Swing, modèle graphique sous forme d'ontologie

*UIOnto* est l'ontologie pour décrire la partie graphique d'une application. Cette ontologie est constituée de deux classes et onze propriétés. Nous en faisons la description détaillée dans l'annexe B section B.1 (page163).

#### MapsUI.rdf

Ci-dessous est présenté un extrait de la description sémantique du modèle de l'interface graphique de l'application "Maps" - "MapsUI.rdf". Cet extrait est la description de l'élément graphique "RouteCalculButton". Cet élément est lié à l'élément logiciel de même nom à travers deux ports "ButtonActionPerformed" permettant de déclencher la calcul du trajet et "GetRouteCalculButton" permettant de récupérer un composant SWING. Ce bouton est situé à droite du conteneur "PositionsContainer" (qui contient les deux entrées textes pour pouvoir renseigner l'adresse de départ et l'adresse d'arrivée).

Une partie de la description sémantique du modèle de l'interface graphique de l'application "Maps" est présente dans l'annexe C section C.1 (page 173).

Listing 6.2 – **Extrait de la description sémantique au format RDF : fichier MapsUi.rdf.part**

```
57      <UIElem rdf:ID="RouteCalculButton">
58  <!-- Décrit l élément graphique permettant de déclencher le calcul de l itinéraire -->
59  <!-- Il est situé à droite du "conteneur" des entrées textes -->
60          <isOnTheRightOf rdf:resource="#PositionsContainer" />
61
62  <!-- Deux liens sont établis avec deux ports du même élément logiciel du modèle ↩
        opérationnel: toujours un lien vers un port permettant de récupérer l élément ↩
        graphique (composant SWING) et le premier lien est vers le port "TRIGGER" permettant ↩
        de déclencher le calcul de l itinéraire -->
63          <ontocompo:uiLinkedWithPort rdf:resource="&ontocompo;-instances#↩
                ButtonActionPerformed" />
64          <ontocompo:uiLinkedWithPort rdf:resource="&ontocompo;-instances#↩
                GetRouteCalculButton" />
65      </UIElem>
```

### 6.2.3 Arbre de tâches descriptif sous forme d'ontologie

L'arbre de tâches de l'application apparaît seulement sous la forme d'une description sémantique et n'est pas "opérationnel" dans notre prototype. Effectivement, aucun mécanisme n'est mis en place tel qu'un réseau de pétri (comme dans [71]) ou un système d'exécution de tâches afin de rendre opérationnel cet arbre.

*TaskOnto* est l'ontologie pour décrire l'arbre de tâches d'une application. Cette ontologie est constituée de six classes et sept propriétés. Nous en faisons la description détaillée dans l'annexe B section B.2 (page165).

#### MapsTasks.rdf

Ci-dessous est présenté un extrait de la description sémantique de l'arbre de tâches de l'application "Maps" - "MapsTasks.rdf". Cet extrait est la description de la tâche "TriggerRouteCalcul". Cette tâche est liée à l'élément logiciel "RouteCalculButton" à travers le port "ButtonActionPerformed". Elle est aussi liée à l'élément graphique "RouteCalculButton". Elle a pour tâche parente la tâche "RetrieveARoute" et en séquence avec la tâche "RouteCalcul".

Une partie de la description sémantique de l'arbre de tâches de l'application "Maps" est présente dans l'annexe C section C.2 (page 175).

Listing 6.3 – **Extrait de la description sémantique au format RDF : fichier MapsTasks.rdf.part**

```
42      <InteractionTask rdf:ID="TriggerRouteCalcul">
43  <!-- Description de la tâche permettant de déclencher le calcul de l itinéraire. -->
44          <hasParentTask rdf:resource="#RetrieveARoute" />
45
46          <isInSequenceWith rdf:resource="#RouteCalcul" />
47
48          <ontocompo:taskLinkedWithPort rdf:resource="&ontocompo;-instances#↩
                ButtonActionPerformed" />
49          <ontocompo:taskLinkedWithUIElem rdf:resource="&uionto;-instances#RouteCalculButton↩
                " />
50      </InteractionTask>
```

### 6.2.4 Liaison entre les modèles

Nous avons définis trois propriétés de nos ontologies afin de lier les modèles entre eux comme défini dans le chapitre 3. La première *taskLinkedWithPort* permet de lier une tâche définie dans TaskOnto à un port d'un élément logiciel. La seconde *uiLinkedWithPort* permet de lier un élément graphique défini dans UIOnto à un port d'un élément logiciel. Et enfin, la dernière *taskLinkedWithUIElem* permet de lier une tâche et un élément graphique. Ces propriétés permettent d'établir des ponts entre les trois modèles et forment les bases aux raisonnements que nous avons sur les liaisons entre les différentes entités constituants les différents points de vue de notre application.

Un exemple de ces liens se situe dans les différentes extractions de descriptions sémantiques présentées ci-dessus et sont reportés sur la figure 6.1 (page 112).

Par exemple, l'élément graphique "RouteCalculButton" (défini à la ligne 57 du listing 6.2) est lié aux ports "ButtonActionPerformed" (défini à la ligne 126 du listing 6.1) et "GetRouteCalculButton"

(défini à la ligne 134 du listing 6.1) de l'élément logiciel "RouteCalculButton" (défini à la ligne 115 du listing 6.1) . Ces liaisons sont visibles respectivement aux lignes 63 et 64 du listing 6.2.

Autre exemple, la tâche "TriggerRouteCalcul" (définie à la ligne 42 du listing 6.3) est liée au port "ButtonActionPerformed" de l'élément logiciel "RouteCalculButton" et à l'élément graphique "RouteCalculButton". Ces liaisons sont visibles respectivement aux lignes 48 et 49 du listing 6.3.

## 6.3 Implémentation des extensions de sélection et des substitutions

Nous présentons dans cette section l'architecture d'OntoCompo ainsi que l'implémentation des différents points de vue d'une application et leur exploitation pour les extensions de sélection et les substitutions.

### 6.3.1 Architecture du prototype OntoCompo

OntoCompo prend son nom de l'ontologie principale décrivant à la fois le modèle opérationnel mais aussi les propriétés pour lier les différents modèles. Ce prototype est conçu en Java. Il manipule des applications, à base de composants Fractal, qui doivent être accompagnées de leur descriptions sémantiques. Afin d'implémenter les différents algorithmes présentés dans ce manuscrit, nous effectuons des requêtes SPARQL, c'est-à-dire des requêtes nous permettant d'interroger la base d'annotations sémantiques apportée par l'application afin de remplir les fonctionnalités nécessaires à la mise en place du processus de composition choisi. Les requêtes SPARQL sont réalisées à l'aide du moteur sémantique CORESE/KGRAM [67].

L'architecture de l'application est représentée sur la figure 6.2. Cette architecture est découpée en six parties. La partie en jaune et accessible à toutes les autres parties est l'ensemble des informations disponibles sur les applications. Ces informations sont l'ensemble des composants fractal et l'ensemble des descriptions sémantiques des applications à fusionner. La partie principale de cette architecture est constituée par l'API. Cette API est en charge de toutes les manipulations à effectuer sur les composants fractals ou descriptions sémantiques pour le déroulement des algorithmes. Les parties en rouges quant à elles représentent les parties spécifiques au processus de composition choisi et constituent aussi l'interface graphique de notre prototype. Une partie chargement est nécessaire afin de récolter et stocker les informations des différentes applications à manipuler. Puis les trois autres parties constituent alors les différentes étapes du processus de composition à savoir "Sélection - Substitution - Placement".

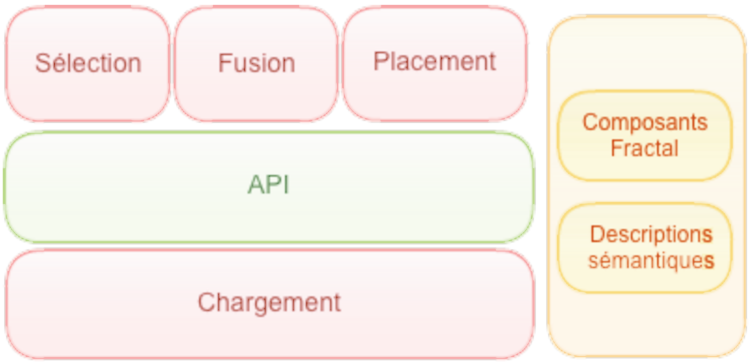



FIGURE 6.2 – Architecture du prototype OntoCompo.

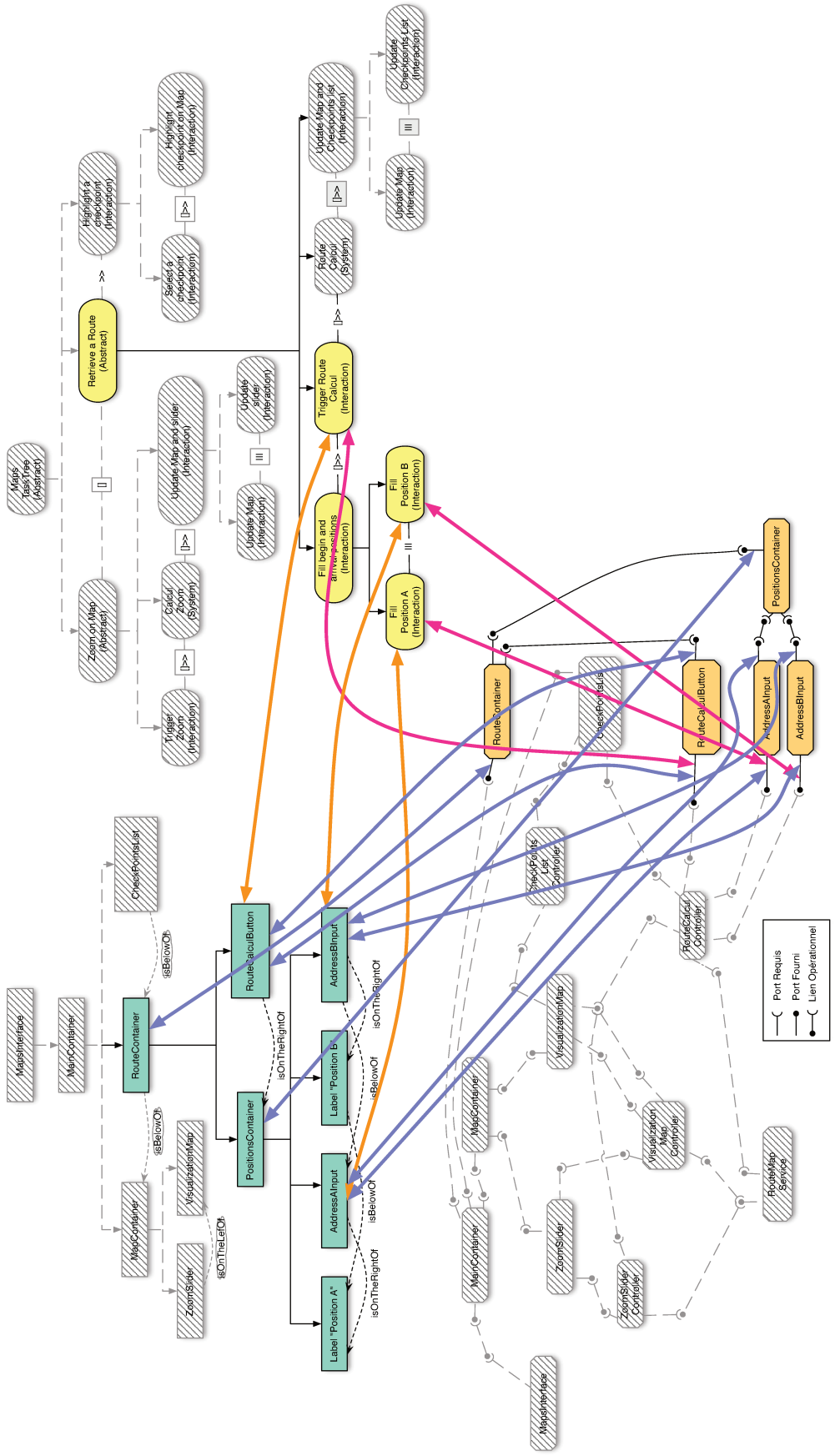



FIGURE 6.1 – Sous-partie des liens entre les modèles de l'application Maps.

Les manipulations des applications s'effectuent à partir de leur interfaces graphiques. Nous utilisons une collection de type Map pour associer un composant SWING à l'identifiant d'un composant fractal. Avec l'utilisation de descriptions sémantiques, chaque entité manipulée a une URI associée. Afin de travailler avec les annotations, nous utilisons une méthode *getAppElemUriFromFractalComponentId* (cf. listing 6.4) qui permet d'obtenir l'URI de l'élément logiciel correspondant à l'identifiant d'un composant fractal. Pour cela, chaque composant fractal met en place un port permettant d'obtenir leur identifiant et chaque élément logiciel de nos descriptions sémantiques contiennent cet identifiant dans leur description. Ainsi, à partir d'un composant SWING sélectionné dans l'interface graphique de l'application, nous pouvons grâce à la collection, obtenir l'identifiant du composant fractal correspondant et obtenir l'URI de l'élément logiciel correspondant dans nos descriptions. Avec les liens entre modèles, nous pouvons alors obtenir facilement les entités liées.

Listing 6.4 – **Code Source Java : méthode getAppElemUriFromFractalComponentId**

```
1  public String getAppElemUriFromFractalComponentId(Application application, String ↩
       fractalComponentId) {
2      try {
3          QueryProcess exec = QueryProcess.create(application.getSemanticGraph());
4
5  /* Requête SPARQL permettant de récupérer l URI de l élément logiciel ayant l identifiant ↩
       passé en paramètre */
6          String query = "SELECT DISTINCT ?appelem WHERE { " +
7                  "GRAPH <" + application.getSemanticInfo().getGraphName() + "> {" +
8                  "?appelem rdf:type <" + OntoCompoProperties.ONTOCOMPO + "#" + ↩
                       OntoCompoProperties.ONTOCOMPO_APPELEMENT_CLASS + "> . " +
9                  "?appelem <" + OntoCompoProperties.ONTOCOMPO + "#" + OntoCompoProperties.↩
                       ONTOCOMPO_HASID_PROPERTY + "> \"" + fractalComponentId + "\"" +
10                 "}" +
11             "}";
12
13         Mappings map = exec.query(query);
14         if(map.size() == 1) {
15             Mapping m = map.get(0);
16             return m.getValue("?appelem").toString();
17         }
18
19         return null;
20     } catch (EngineException e) {
21         e.printStackTrace();
22         return null;
23     }
24 }
```

### 6.3.2 Implémentation des extensions de sélections

La manipulation des différents modèles sont effectués grâce à des requêtes sémantiques effectuées sur les descriptions des applications à composer. Pour cela, nous utilisons le langage SPARQL qui permet d'effectuer des requêtes dans un format proche des requêtes SQL pour les bases de données relationnelles. Ainsi, des requêtes sémantiques permettent de réaliser les algorithmes de l'extension. Par exemple, l'extension suivant la tâche parente à partir de la sélection d'un élément graphique est défini par la méthode *extTaskParentFromUI* (cf. listing 6.5) mettant en place l'algorithme de parcours des modèles de l'application nécessaire à l'extension et effectuant plusieurs requêtes SPARQL pour obtenir les informations voulues.

Listing 6.5 – **Code Source Java : méthode extTaskParentFromUI**

```
1  public Selection extTaskParentFromUI(Application application, ArrayList<String> ↩
       fractalComponentIds, int deep) {
2  /* On appelle "completionUI" pour récupérer les tâches liées aux éléments graphiques ↩
       sélectionnés */
3      Selection resultSel = this.completionUI(application, fractalComponentIds);
4
5  /* Pour chaque tâche trouvée, on effectue l extension par la tâche parente */
6      for(String taskUri : resultSel.getTasksUri().get(application)) {
7          resultSel = resultSel.union(this.extTaskParent(application, taskUri, resultSel, ↩
               deep));
8      }
9
10 /* On appelle "completionTask" pour récupérer les éléments graphiques liés aux nouvelles ↩
       tâches */
11     resultSel = resultSel.union(this.completionTask(application, resultSel.getTasksUri().↩
           get(application)0));
12
13     return resultSel;
14 }
15
16 /* Méthode permettant d effectuant l extension suivant la tâche parente, avec une ↩
       profondeur donnée */
17 public Selection extTaskParent(Application application, String taskUri, Selection sel, int↩
        deep) {
18     if(deep == 0) {
19         return sel;
20     }
21
22     if(deep == 1) {
23         return this.extTaskParent(application, taskUri, sel);
24     }
25
26     if(deep > 1) {
27         int newDeep = 0;
28
29         String parentTaskUri = taskUri;
30
31         while(parentTaskUri != null || newDeep!=(deep-1)) {
32             parentTaskUri = this.ocsa.getTaskParentUri(application, parentTaskUri);
33             newDeep++;
34         }
35
36         Selection extSel = this.extTaskParent(application, taskUri, sel, newDeep);
37
38         return this.extTaskParent(application, parentTaskUri, extSel);
39     }
40
41     return null;
42 }
```

### 6.3.3 Implémentation des substitutions

L'implémentation des substitutions utilise une méthode au coeur de l'étape de substitution qui est la vérification qu'un port est "substitutable" par un autre. Les conditions de substitutions sont décrites dans le chapitre 5. Son implémentation est effectuée par la méthode *isSubtitutableBy* (cf. listing 6.6) définit par le code suivant. Ce code est accompagné des requêtes SPARQL permettant de récupérer le rôle d'un port, le type d'un port et de savoir si le port est fourni.

Listing 6.6 – **Code Source Java : méthode isSubtitutableBy**

```
1  public boolean isSubtitutableBy(Application application1, Application application2, String portApp1Uri, String portApp2Uri) {
2      if(this.isProvidedPort(application1, portApp1Uri) && this.isProvidedPort(application2,portApp2Uri)) {
3          String portApp2Role = this.getPortRole(application2, portApp2Uri); // Rôle du port à substituer
4  /* Si le port à substituer a le rôle "UIComponent" alors la substitution est possible si le port gardé a aussi ce rôle. */
5          if(portApp2Role.equals("<" + OntoCompoProperties.ONTOCOMPO_INSTANCES + "#" + OntoCompoProperties.ONTOCOMPO_UICOMPONENTROLE_INSTANCE + ">")) {
6              return true;
7          }
8
9          String portApp2Type = this.getPortType(application2, portApp2Uri); // Type du port à substituer
10         String portApp1Type = this.getPortType(application1, portApp1Uri); // Type du port à garder
11
12 /* Si le port à substituer est de type "INPUT" alors la substitution est possible dans les cas où le port à garder est de type "INPUT" ou "OUTPUT". */
13         if(portApp2Type.equals("<" + OntoCompoProperties.ONTOCOMPO_INSTANCES + "#" + OntoCompoProperties.ONTOCOMPO_INPUT_INSTANCE + ">")) {
14             return portApp1Type.equals("<" + OntoCompoProperties.ONTOCOMPO_INSTANCES + "#" + OntoCompoProperties.ONTOCOMPO_INPUT_INSTANCE + ">") ||
15                 portApp1Type.equals("<" + OntoCompoProperties.ONTOCOMPO_INSTANCES + "#" + OntoCompoProperties.ONTOCOMPO_OUTPUT_INSTANCE + ">");
16         }
17
18 /* Si le port à substituer est de type "OUTPUT" alors la substitution est possible dans les cas où le port à garder est de type "OUTPUT". */
19         if(portApp2Type.equals("<" + OntoCompoProperties.ONTOCOMPO_INSTANCES + "#" + OntoCompoProperties.ONTOCOMPO_OUTPUT_INSTANCE + ">")) {
20             return portApp1Type.equals("<" + OntoCompoProperties.ONTOCOMPO_INSTANCES + "#" + OntoCompoProperties.ONTOCOMPO_OUTPUT_INSTANCE + ">");
21         }
22
23 /* Si le port à substituer est de type "TRIGGER" alors la substitution est possible dans tous les cas. */
24         if(portApp2Type.equals("<" + OntoCompoProperties.ONTOCOMPO_INSTANCES + "#" + OntoCompoProperties.ONTOCOMPO_TRIGGER_INSTANCE + ">")) {
25             return true;
26         }
27
28     }
29
30     return false;
31 }
32
33
34 /*************/
35
36 /* Requête SPARQL utilisée par la méthode "isProvidedPort" */
37 String query = "ASK { " +
38         "GRAPH <" + application.getSemanticInfo().getGraphName() + "> {" +
39             portUri + " rdf:type <" + OntoCompoProperties.ONTOCOMPO + "#" + OntoCompoProperties.ONTOCOMPO_PROVIDEDPORT_CLASS + "> . " +
40         "}" +
41     "}";
42
43 /* Requête SPARQL utilisée par la méthode "getPortType" */
44 String query = "SELECT ?portType WHERE { " +
45         "GRAPH <" + application.getSemanticInfo().getGraphName() + "> {" +
46             portUri + " <" + OntoCompoProperties.ONTOCOMPO + "#" + OntoCompoProperties.ONTOCOMPO_HASTYPE_PROPERTY + "> ?portType " +
47         "}" +
```

```
48      "}";
49
50  /* Requête SPARQL utilisée par la méthode "getPortRole" */
51  String query = "SELECT ?portRole WHERE { " +
52          "GRAPH <" + application.getSemanticInfo().getGraphName() + "> {" +
53              portUri + " <" + OntoCompoProperties.ONTOCOMPO + "#" + OntoCompoProperties.↩
                    ONTOCOMPO_HASROLE_PROPERTY + "> ?portRole " +
54          "}" +
55      "}";
```

La seconde méthode principalement utilisée lors des substitutions est la méthode permettant de générer le code correspondant à l'Adapteur. La logique de cette création est implémentée grâce à la méthode *substPorts* (cf. listing 6.7) qui permet de générer un Adapteur comme celui présenté dans l'annexe E (page 183). Cette méthode s'appuie grandement sur l'introspection que nous pouvons effectuer sur les composants fractal sans pour autant avoir accès au code source de ceux-ci.

Listing 6.7 – **Code Source Java : méthode substPorts**

```
1   public boolean substPorts(Application application1, Application application2, String ↩
        portApp1Uri, String portApp2Uri) {
2   /* ... */
3
4   /* Récupération des composants fractal associés aux ports à remplacer et à garder */
5
6   /* ... */
7                       String requiredInterfaceName = "";
8                       String requiredInterfaceUri="";
9                       try {
10                          BindingController bindingitf = (BindingController)↩
                                toKeepConnectedRequiredComponent.getFcInterface("binding-↩
                                controller");
11                          String[] interfaces = bindingitf.listFc();
12
13                          if(interfaces.length != 0) {
14
15                              for(String i: interfaces) {
16                                  Object o = bindingitf.lookupFc(i);
17                                  Interface itfserver = (Interface) o;
18
19                                  if(itfserver.getFcItfName().equals(toKeepPortName)) {
20                                      Interface requiredInterface = (Interface) ↩
                                            toKeepConnectedRequiredComponent.getFcInterface(i)↩
                                            ;
21
22                                      requiredInterfaceName = (requiredInterface.↩
                                            getFcItfType().toString().split("/"))[0];
23                                      requiredInterfaceUri = (requiredInterface.getFcItfType↩
                                            ().toString().split("/"))[1];
24                                  }
25                              }
26                          }
27                      } catch (NoSuchInterfaceException e) {
28                          e.printStackTrace();
29                      }
30
31                      String toKeepPortInterfaceUri = null;
32                      String toRemovePortInterfaceUri = null;
33
34  /* Récupération de la signature des interfaces JAVA associées au port à garder */
35                      for(Object objBis : toKeepFractalComponent.getFcInterfaces()) {
36                          Interface cserveriBis = (Interface) objBis;
37
```

```
38                          String controlInterfaceName = (cserveriBis.getFcItfType().toString↩
                                ().split("/"))[0];
39                          String controlInterfaceUri = (cserveriBis.getFcItfType().toString↩
                                ().split("/"))[1];
40
41
42                          if(controlInterfaceName.equals(toKeepPortName)) {
43                              toKeepPortInterfaceUri = controlInterfaceUri;
44                              break;
45                          }
46                      }
47
48  /* Récupération de la signature des interfaces JAVA associées au port à supprimer */
49                      for(Object objBis : toRemoveFractalComponent.getFcInterfaces()) {
50                          Interface cserveriBis = (Interface) objBis;
51
52                          String controlInterfaceName = (cserveriBis.getFcItfType().toString↩
                                ().split("/"))[0];
53                          String controlInterfaceUri = (cserveriBis.getFcItfType().toString↩
                                ().split("/"))[1];
54
55                          if(controlInterfaceName.equals(toRemovePortName)) {
56                              toRemovePortInterfaceUri = controlInterfaceUri;
57                              break;
58                          }
59                      }
60
61  /* Récupération des méthodes JAVA (correspondants aux interfaces JAVA) que doit ↩
        implémenter l Adapteur */
62                      String methodsToImplement=this.getMethodsToImplement(↩
                            toKeepPortInterfaceUri);
63                      methodsToImplement += this.getMethodsToImplement(↩
                            toRemovePortInterfaceUri);
64
65  /* Génération du fichier définissant l Adapteur */
66                      String generatedFileContent = "";
67                      generatedFileContent += "package ontocompo.compo." + application1.↩
                            getName().toLowerCase() + application2.getName().toLowerCase() + "↩
                            ;\n" +
68
69                          "import org.objectweb.fractal.fraclet.annotations.Component;\n" +
70                          "import org.objectweb.fractal.fraclet.annotations.Interface;\n" +
71                          "import org.objectweb.fractal.fraclet.annotations.Requires;\n" +
72                          "\n" +
73                          "@Component(provides={" +
74
75                          "@Interface(name=\"" + toKeepPortName + "\",signature=" + ↩
                                toKeepPortInterfaceUri + ".class)," +
76                          "@Interface(name=\"" + toRemovePortName + "\",signature=" + ↩
                                toRemovePortInterfaceUri + ".class)}) //Interface fourni par "↩
                                 + application1.getName() + ": port GetInput, Interface fourni↩
                                 par " + application2.getName() + ": GetInputFrom\n" +
77                          "\n" +
78                          "public class Adapter" + application1.getName() + toKeepPortLabel ↩
                                + application2.getName() + toRemovePortLabel + " extends ↩
                                ontocompo.adapter.lib.Adapter implements ontocompo.api.↩
                                InitComponentItf, " +
79                          toKeepPortInterfaceUri + ", " + toRemovePortInterfaceUri +
80                          "{\n" +
81
82                              "\t@Requires(name=\"" + requiredInterfaceName + "\")\n" +
83                              "\t" + requiredInterfaceUri + " " + requiredInterfaceName + ";↩
                                     // Same port as port connected to keep port: "+ ↩
                                    application1.getName() + ":" + toKeepPortName + "\n" +
84
85
```

```
86                              "\tpublic Adapter" + application1.getName() + toKeepPortLabel ↩
                                    + application2.getName() + toRemovePortLabel + "() {\n" +
87                                  "\t\t// TODO Auto-generated constructor stub\n" +
88                              "\t}\n" +
89
90                              methodsToImplement +
91
92                              "\tpublic void init() {\n" +
93                                  "\t\t// Nothing TODO!\n" +
94                              "\t}\n" +
95                              "\n" +
96                              "\tpublic String getUid() {\n" +
97                                  "\t\treturn ontocompo.compo." + application1.getName().↩
                                        toLowerCase() + application2.getName().toLowerCase() +↩
                                         ".Adapter" + application1.getName() + toKeepPortLabel↩
                                         + application2.getName() + toRemovePortLabel + ";\n" ↩
                                        +
98                              "\t}\n" +
99                              "\n" +
100                         "}\n";
101
102                     Tools.writeGeneratedFile("ontocompo" + File.separator + "compo" + File↩
                            .separator + application1.getName().toLowerCase() + application2.↩
                            getName().toLowerCase() + File.separator + "Adapter" + ↩
                            application1.getName() + toKeepPortLabel + application2.getName() ↩
                            + toRemovePortLabel + ".java", generatedFileContent);
103
104 /* ... */
105
106 /* Fin du traitemant pour la méthode */
107
108 /* ... */
109 }
```

Enfin, les dernières étapes pour effectuer la substitution est de générer les connexions entre composants fractals afin d'obtenir comme résultat de la composition une application opérationnelle.

## 6.4 OntoCompo : une composition dirigée par la manipulation des interfaces graphiques

Nous présentons dans cette section l'interface graphique d'OntoCompo pour les différentes étapes du processus de composition.

### 6.4.1 Chargement des applications

L'écran de chargement des applications (cf. figure 6.3) est constitué d'un menu permettant ce chargement et d'une partie centrale permettant de manipuler les applications une fois chargées. Cette partie permet ainsi d'interagir avec les applications afin de comprendre leur fonctionnement et les différentes fonctionnalités qu'elles proposent.

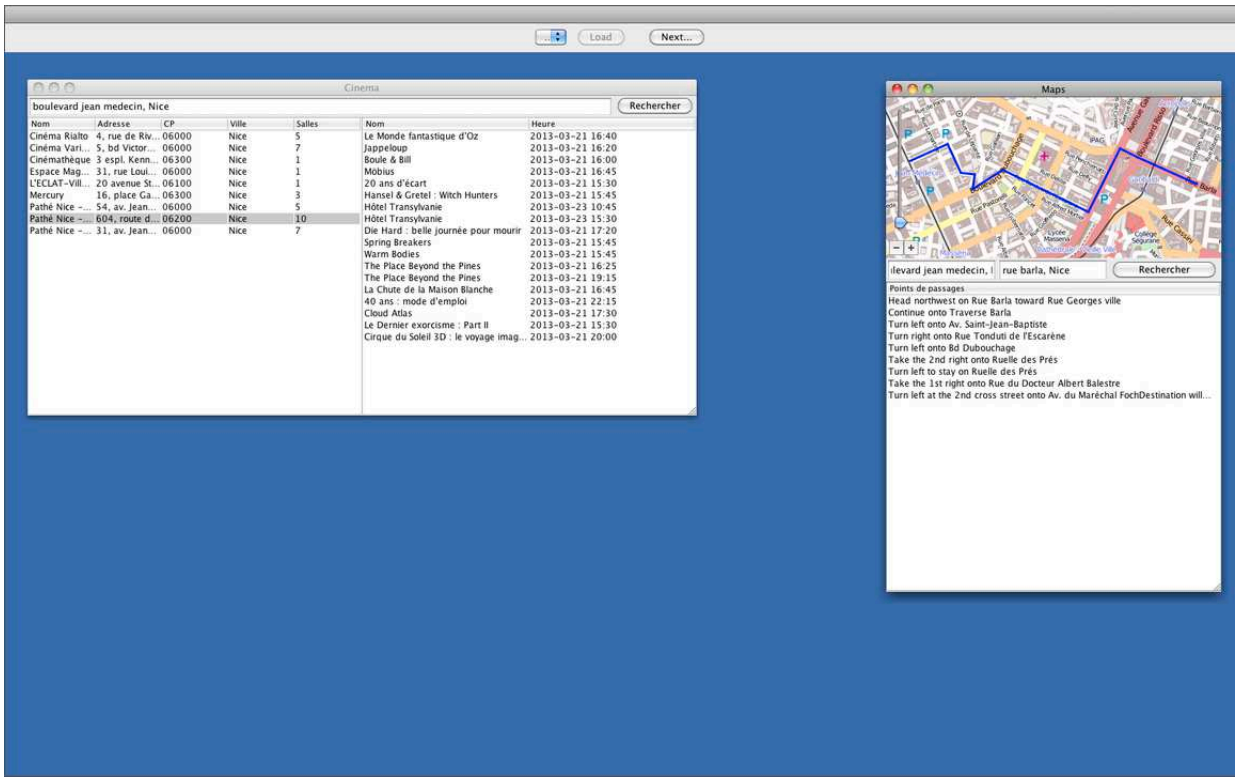



FIGURE 6.3 – Ecran de chargement d'OntoCompo.

### 6.4.2 Sélection

L'écran de sélection (cf. figure 6.4) reprend la partie centrale où sont chargées les applications. Leur fonctionnement initial est alors inhibé afin de pouvoir ajouter les différentes interactions nécessaires à la sélection. La sélection peut s'effectuer directement en cliquant sur un élément qui est alors encadré en orange. Les extensions de sélection peuvent être combinées en les sélectionnant une par une dans le menu présent au dessus de la partie centrale. La profondeur d'exploration peut être définie pour chaque extension. Une fois les éléments sélectionnés, ils peuvent être ajoutés à la sélection, leur encadrement devient vert.

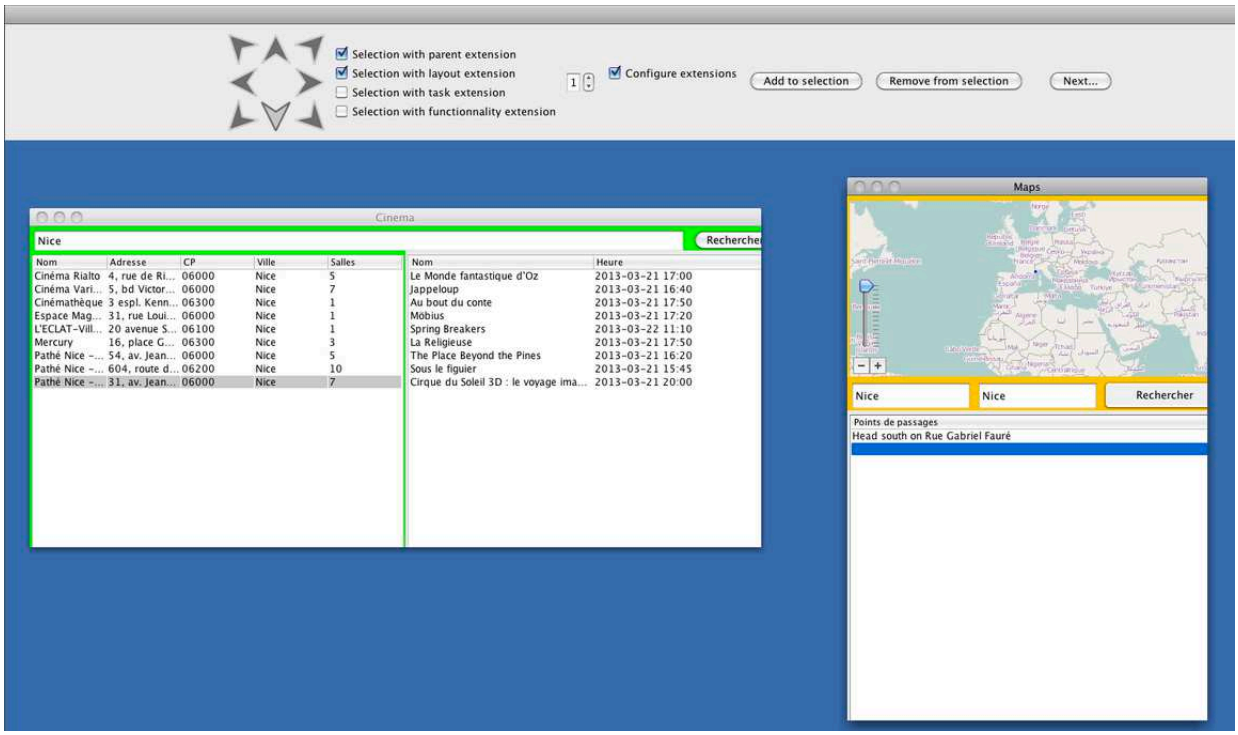



FIGURE 6.4 – Ecran de sélection d'OntoCompo.

### 6.4.3 Substitutions

L'écran de substitutions (cf. figure 6.5) reprend la partie centrale où sont chargées les applications et grise les éléments graphiques qui ne sont pas sélectionnés. La substitution s'effectue entre 2 éléments graphiques, le premier devant être déclaré comme "gardé" (encadrement en vert) et le second devant être déclaré comme "supprimé" (encadrement en rouge). Une fois les 2 éléments sélectionnés, la substitution peut être effectuée en cliquant sur le bouton "Fusion". C'est à ce moment là que le code des adapteurs produits par la substitution est généré.

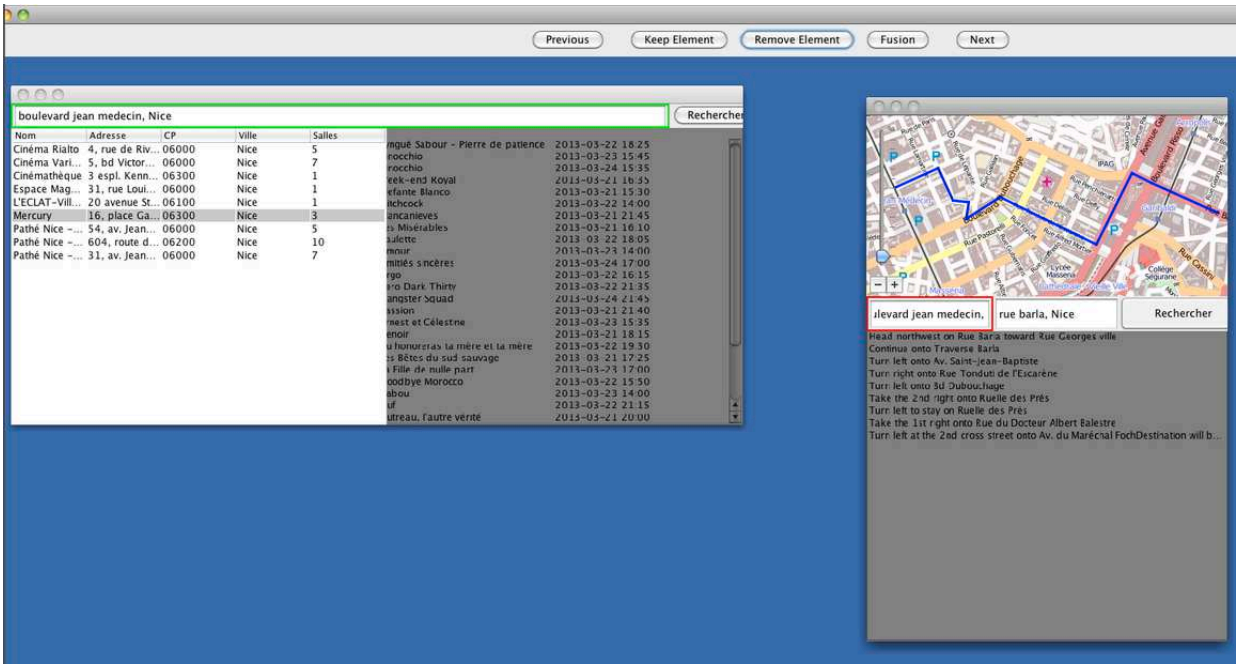



FIGURE 6.5 – Ecran de substitutions d'OntoCompo.

### 6.4.4 Placement

L'écran de placement (cf. figure 6.6) modifie la partie centrale pour faire apparaître une fenêtre contenant les éléments graphiques restants. L'utilisateur peut alors placer les éléments dans cette fenêtre par glisser/déposer.

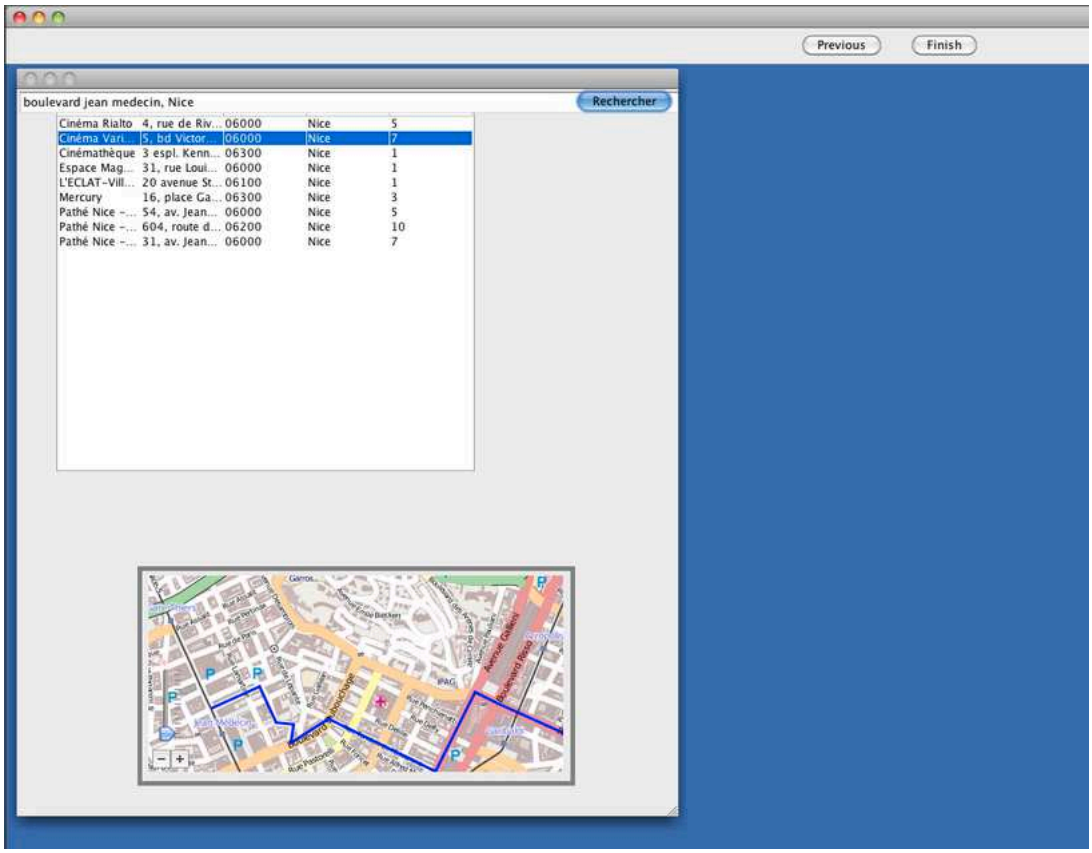



FIGURE 6.6 – Ecran de placement d'OntoCompo.

## 6.5 Conclusion

Ce prototype propose une première implémentation des modèles et algorithmes présentés dans ce manuscrit. Toutes les extensions n'ont pas été traitées lors du développement du prototype. Effectivement, le développement de ce prototype est centré autour du choix du processus de composition effectué *Sélection - Substitutions - Placement* afin de mettre en place des expérimentations utilisateurs nous permettant d'obtenir des informations sur les besoins des développeurs lors de leurs démarches de compositions d'applications. Ce développement a permis de mettre en œuvre les modèles des différents points de vue d'une application à travers la mise en place de différentes ontologies et l'utilisation de composants fractal comme base de développement et de manipulation des applications. Les substitutions s'appuient notamment sur l'introspection possible de ces composants. Les requêtes SPARQL effectuées tout au long du processus suivent les algorithmes que nous avons définis dans ce manuscrit. Ainsi, l'arbre de tâches de l'application sert de base descriptive de l'application mais n'est en aucun cas manipulé. L'application résultante de la composition est alors opérationnelle mais ne peut pas être directement composer de nouveau avec une autre application, une intervention du développeur étant nécessaire afin de re-construire un arbre de tâches cohérent pour l'application résultante.

Dans le chapitre suivant, nous présentons les résultats des expérimentations faites à l'aide du prototype OntoCompo.

# Publications

Ce prototype a été présenté plus particulièrement dans les publications suivantes :

## Publications dans des Conférences Internationales

[BPDRR11] Christian Brel, Anne-Marie Pinna-Déry, Philippe Renevier, and Michel Riveill. OntoCompo : A Tool To Enhance Application Composition. In *13th IFIP TC13 Conference in Human-Computer Interaction INTERACT 2011 (Interact 2011)*, pages 588–591, September 2011.

## Publications dans des Conférences Nationales

[BR11] Christian Brel and Philippe Renevier. Composing Applications with OntoCompo. In *Conference Interaction Homme-Machine (IHM)*, Nice, October 2011.

CHAPITRE 7

# Validation de l'approche par les utilisateurs

**Sommaire**



NOUS présentons dans ce chapitre la mise en place et les résultats d'une série d'expérimentations menée avec des utilisateurs ayant un profil d'**informaticien-développeur**. Le but de ces tests utilisateurs est de valider l'approche proposée dans cette thèse et notamment la composition d'applications dirigée par la manipulation de leurs interfaces graphiques.

## 7.1 Tour d'horizon des méthodes d'évaluations

Si nous prenons la classification usuelle des méthodes d'évaluations (cf. figure 7.1) des interfaces utilisateurs [65, 66], nous constatons que nous souhaitons effectuer une évaluation à base d'une technique expérimentale qui est l'utilisation du prototype réalisé. Les ressources matérielles nécessaires pour ce type d'évaluation (cf. figure 7.2) se situent entre la papier/crayon et l'utilisation d'un "labo d'ergonomie" c'est-à-dire des outils permettant la capture vidéo, audio des manipulations effectuées par l'utilisateur. Ceci est confirmé par le tableau récapitulatif de la dimension de la présence de l'utilisateur et de l'implémentation de l'interface (tiré de [65]) qui préconise, dans le cas d'une interface implémentée avec la présence d'un utilisateur, de capter les actions utilisateurs à l'aide de matériel audiovisuel. Nous avons appliqué (cf. section 7.4) cette préconisation dans nos tests.

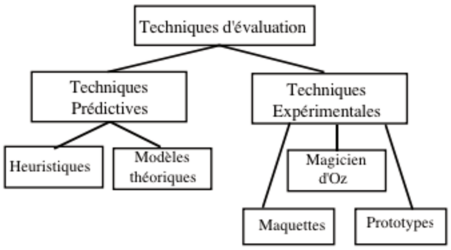



FIGURE 7.1 – Classification usuelle des méthodes d'évaluations des interfaces utilisateurs tiré de [65, 66]

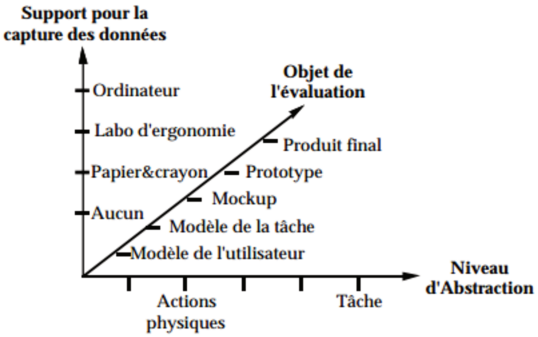



FIGURE 7.2 – Sous-espace des ressources matérielles défini dans [65]

| \ Utilisateur Interface \ | Absent | Présent |
|---|---|---|
| Non implémentée | Evaluation de spécifications externes, de simulations papiers. Outils formels d'évaluation (grammaires, cognitive walkthrough, etc.) | Questions à l'utilisateur (interview, enquêtes, etc.), simulations par Magicien d'Oz |
| Implémentée | Evaluation de prototype, de simulation de l'interface et/ou de simulation de l'utilisateur | Observation de l'utilisateur par le biais de l'ordinateur (trace des actions essentiellement), ou en train d'utiliser ce dernier (audiovisuel) |

FIGURE 7.3 – Tableau récapitulatif de la dimension de la présence de l'utilisateur et de l'implémentation de l'interface tiré de [65]

Dans les travaux précédents, autour de la composition d'interface utilisant le langage UsiXML, les auteurs ont effectué une évaluation [51] à base de la méthode GOMS afin d'établir la plus-value en terme de gain de temps lors de l'utilisation de leur outil. Dans notre cas, nous ne pouvons effectuer le même type d'évaluation. D'une part, l'objectif de ces tests est avant tout d'évaluer la compréhension du processus de composition et l'intérêt des liens entre les modèles de l'application pour effectuer la composition. L'objectif du gain de temps comparé aux méthodes manuelles (manipulation de code) est secondaire même si celui-ci peut devenir pertinent une fois le process validé. D'autre part, la méthode GOMS nécessite de pouvoir décrire les buts et les sous-buts. Or, dans l'étape de sélection, nous avons le but principal qui est de pouvoir sélectionner les éléments à composer, mais nous ne

pouvons décomposer ce but car nous ne voulons pas imposer l'utilisation d'une des aides disponibles (extensions de sélection) à l'utilisateur.

## 7.2 Hypothèses

Il s'agit à travers le protocole de tests (voir ci-après) d'évaluer la véracité de l'hypothèse suivante : le seul accès aux éléments des interfaces graphiques des applications suffit à réaliser la composition grâce aux modèles sous-jacents. En cas de remise en question de cette hypothèse, nous émettons l'hypothèse complémentaire que notre description des applications contient les informations supplémentaires requises pour effectuer néanmoins une composition valide. Une hypothèse complémentaire est de considérer que le processus de composition est compréhensible et réalisable.

## 7.3 Objectifs

Les objectifs sont donc d'évaluer :
- la **compréhension** par les développeurs des différentes étapes du **processus de composition** d'applications Sélection - Substitution - Placement à partir de la manipulation des éléments graphiques,
- leur **aptitude à effectuer** ce processus à l'aide du prototype développé et plus exactement d'évaluer si l'interface graphique de cet outil permet aux développeurs de réaliser et comprendre facilement ce processus,
- si les **informations fournies** (accès seulement aux interfaces graphiques des applications et aux opérations de manipulation) suffisent à comprendre et réaliser le processus de composition (et dans le cas contraire, quelles informations leur auraient été nécessaires),
- la **compréhension et la pertinence des extensions** de sélection, aides accessibles lors de l'étape de sélection et plus particulièrement, **l'apport des tâches et des liaisons** entre celles-ci et les autres modèles pour décrire l'application durant le processus de composition.

## 7.4 Organisation des tests

L'utilisateur est placé face à l'écran qui permet d'utiliser le prototype développé. Son interface permet de composer uniquement les applications sélectionnées. Toutes les manipulations effectuées sont enregistrés afin de pouvoir les analyser grâce à une capture vidéo de l'écran. De même, les échanges oraux entre l'utilisateur et le testeur sont enregistrés. Au fur et à mesure du test, des documents imprimés portant sur le code généré ou sur des informations complémentaires (modèle de tâches, de composants) sont proposés à l'utilisateur. Une caméra est placée au dessus de l'écran pour filmer ce que désignera l'utilisateur sur ces documents. Le testeur est placé à droite de l'utilisateur et un observateur est placé sur sa gauche ou derrière l'utilisateur et prend des notes.

Les deux applications à composer sont celles présentées dans l'étude de cas (cf. chapitre 1 section 1.4), à savoir :
- l'application "Cinema" qui permet à partir d'une adresse donnée, d'obtenir la liste des cinémas les plus proches, puis lorsqu'un cinéma est sélectionné dans la liste des réponses, la liste des séances qui y sont programmées est affichée ;

– l'application "Maps" qui permet d'obtenir l'itinéraire pour se rendre d'une adresse de départ à une adresse d'arrivée en l'affichant sur une carte et en listant les différents points caractéristiques de l'itinéraire.

Le but de la composition à réaliser est d'obtenir une application qui permet d'obtenir la liste des cinémas les plus proches d'une adresse donnée, puis lors de la sélection d'un cinéma, d'afficher l'itinéraire pour s'y rendre, sur la carte sans la liste des points caractéristiques.

Pour effectuer une comparaison, nous avons établi la liste des actions qu'il nous semble pertinent d'effectuer pour réaliser le scénario de composition demandé (cf. annexe D ).

Ces tests se sont déroulés après une phase de calibrage auprès de deux pré-testeurs, qui nous ont permis d'ajuster la procédure de test. Ceci nous a conduit à élaborer le déroulement présenté dans la section suivante.

## 7.5 Déroulement du test

Chaque entretien dure environ une heure. Il se déroule suivant les étapes suivantes présentées dans l'annexe F (page 185).

La démarche suivi durant ce test est d'alterner les phases de familiarisation (l'utilisateur manipule l'interface et les différentes actions réalisables avec le prototype) et les phases de réalisation d'une tâche demandée. Cette démarche se termine par un débriefing sous la forme d'un entretien (cf. annexe G page 189). Cette démarche nous permet de garantir un certain contrôle sur le déroulement du test et notamment sur le fait que nous ne voulons pas évaluer principalement l'interface graphique du prototype, des explications sur l'utilisation de l'interface du prototype étant librement donné par le testeur sous la demande de l'utilisateur tout au long du test.

## 7.6 Difficultés du scénario

Bien que le scénario de composition utilisé lors de ces tests soit simple, nous avons identifié trois difficultés particulières à observer. Le but de ces difficultés est de rapprocher l'utilisateur le plus possible d'un cas réel d'utilisation de notre outil et de pouvoir observer sa réaction face à ces difficultés.

### Similarité trompeuse de deux éléments graphiques

Tout d'abord, les deux boutons présents dans les deux applications ont la même forme et ont le même intitulé "Search". Nous pensons que cela risque d'induire en "erreur" l'utilisateur en l'incitant à substituer ces deux boutons alors que la composition visée ne le demande pas.
Ceci permet de tester la suffisance des informations fournies à l'utilisateur et notamment l'influence que peuvent avoir les arbres de tâches des applications. En effet, les tâches montrent dans ce cas que le premier bouton permet d'obtenir la liste des cinémas et que le second bouton sera à relier avec la sélection d'un cinéma dans la liste.

### Substitution de deux éléments graphiques hétérogènes

La seconde difficulté réside dans le fait de devoir substituer deux éléments de natures différentes. Le premier, celui à conserver, est une liste (la liste des cinémas les plus proches de l'adresse renseigné dans l'interface graphique de l'application "Cinema"). Le second, à remplacer, est une entrée texte (l'entrée texte permettant de renseigner l'adresse d'arrivée nécessaire au calcul de l'itinéraire dans l'interface graphique de l'application "Maps"). La substitution est possible car l'élément logiciel associé à la liste fournit un port de type $INPUT$ pour obtenir le cinéma sélectionné dans la liste.
Cette substitution nous permettant d'exprimer le fait que nous voulons automatiquement renseigner l'adresse d'arriver de l'itinéraire avec l'adresse du cinéma sélectionné, peut perturber l'utilisateur puisque la sélection de la colonne "adresse" présent dans la liste de cinémas ne peut être directement sélectionnée. Ceci nous permet de vérifier si la vue composant peut apporter des aides notamment à travers les fonctionnalités fournies ou requises.

### Génération de code avec deux méthodes de même signature

Enfin la dernière difficulté vient du code présenté à l'utilisateur après une substitution. Il indique deux fois la même signature de méthode à compléter. L'adapteur généré par la substitution doit comporter plusieurs ports. Or un port fourni correspond à une implémentation d'une interface logicielle. Il se trouve que parmi les deux ports fournis que doit avoir l'adapteur, il y a deux fois la même signature de méthode *public String getInput*() (cf. annexe E page 183). Même si cette difficulté vient plutôt de l'implémentation avec le langage Java qui ne permet pas de faire la différence entre les deux méthodes de même signature provenant de deux interfaces logicielles différentes, nous nous attendons à ce que l'utilisateur sache comment réagir. En effet dans cet exemple, les deux méthodes *getInput*() doivent produire le même résultat : la fusion des deux est possible.

## 7.7 Les participants

Nous avons effectué ce test avec neuf participants séparés en deux groupes. Le premier groupe contient quatre utilisateurs n'ayant jamais manipulé d'outils de composition d'applications. Le second groupe contient cinq utilisateurs qui ont déjà utilisé au moins un outil pour effectuer de la composition mais pas forcément entre applications : composition de services, composition d'interfaces graphiques ... Nous nous attendions à percevoir une différence de comportement entre les deux groupes d'utilisateurs. Mais nous n'avons pas observé de différence significative. Ceci est peut-être dû au petit nombre de participants. Par contre, les participants "experts" ont eu un comportement inattendu :

- Certains participants "experts" ont exprimé de la méfiance vis à vis des outils génératifs, redoutant un résultat non modifiable.
- Globalement, les participants "experts" ont manifesté plus de curiosité, voulant compléter leurs connaissances afin de déterminer les mécanismes sous-jacents qu'ils ne pouvaient inférer.

## 7.8 Données recueillies

Nous présentons ici les principales informations récoltées pendant les entretiens et qui concernent :
- le ressenti des participants vis à vis du processus de composition,
- les informations complémentaires demandées,

– l'apport des tâches dans le processus de composition et
– les préférences des participants vis à vis des extensions.

A travers l'observation des actions des participants, nous avons déterminé :

– l'utilisation des différentes extensions proposées et
– le comportement des participants vis à vis des difficultés relatives du scénario.

Le débriefing final, basé sur le questionnaire présenté dans l'annexe G, nous a permis de mieux cerner la compréhension que les participants ont eu du processus de compositions et des extensions à utiliser.

## 7.9 Résultats des expérimentations

Cette section présente l'analyse des données collectées.

### 7.9.1 Concernant le processus

22% (2/9) des utilisateurs auraient voulu avoir l'étape de Sélection et l'étape de Substitutions regroupées en une seule et même étape afin de sélectionner et composer dans la foulée les différents éléments. Les extensions seraient effectivement dans ce cas une aide à la compréhension du fonctionnement des applications à composer.

Lors de l'étape de placement, tous les utilisateurs ont demandé à avoir aussi la possibilité de conserver les éléments tels qu'ils étaient positionnés dans leur interface d'origine, surtout si l'extension par mise en page (layout) est utilisée.

### 7.9.2 Concernant la sélection

Le tableau 7.1 résume l'utilisation des extensions. Le test comportant plusieurs sélections, les utilisateurs ont variés dans l'emploi des extensions. Ils n'ont pas utilisé la ou les mêmes extensions dans les différentes sélections à réaliser. Ils ont fait parfois au moins une sélection sans extension, souvent avec une extension, parfois avec deux. Les utilisateurs ayant parfois utilisé plusieurs extensions, le cumul est supérieur à 100% (cf. tableau 7.2).

| Type d'Extension | Utilisation | Besoin d'informations supplémentaires |
|---|---|---|
| **Mise en page** | 44% (4/9) | Non, pour aucun |
| **Tâches** | 67% (6/9) | Oui, pour 83% d'entre eux (5/6) |
| **Eléments logiciels** | 11% (1/9) | Oui |

TABLE 7.1 – Utilisation des extensions lors du test utilisateur.

#### Extension "mise en page"

44% (4/9) des utilisateurs se sont principalement basés sur les extensions s'appuyant sur la mise en page des éléments des interfaces graphiques, avec, pour tous ces utilisateurs, une nette préférence pour l'extension par le conteneur parent (extension qui permet d'étendre la sélection à tous les éléments étant contenu dans le même conteneur graphique que l'élément sélectionné).

| Réalisation d'au moins une sélection avec | | | |
|---|---|---|---|
| **0 extension (dans le cas où l'extension utilisée ne convenait plus)** | **1 extension** | **2 extensions** | **3 extensions** |
| 33% (3/9) | 78% (7/9) | 22% (2/9) | 0% |

TABLE 7.2 – Proportions d'utilisation des extensions.

**Extension "tâches"**

67% (6/9) d'entre eux se sont plutôt basés sur les extensions s'appuyant sur l'arbre de tâches. Seulement 22% (2/9) connaissaient la notion d'arbre de tâches avant le test.

**Extension "éléments logiciels"**

Enfin un des utilisateurs a utilisé les extensions s'appuyant sur les liens opérationnels (liens entre les éléments logiciels).

Il est à noter que pour 44% (4/9) des utilisateurs, une utilisation combinée des extensions, s'appuyant notamment sur les tâches et sur les liens opérationnels, leur semblerait plus adéquate. Par exemple, étendre la sélection par les liens opérationnels tout en conservant seulement les éléments impliqués dans une même tâche, ou pour sélectionner la tâche à conserver quand l'élément graphique est impliqué dans plusieurs tâches.

La répartition de l'utilisation des extensions durant la phase de manipulation du test est à mettre en relation avec l'expression effectuée par les utilisateurs durant la phase d'interview du test, sur les informations dont ils auraient besoin en supplément des interfaces graphiques des applications pour mener à bien la composition.

### 7.9.3 Concernant les informations complémentaires et apport des tâches

Un analyse générale permet d'affirmer que des informations complémentaires rendraient le processus plus facile. Nous faisons à titre de comparaison un tableau récapitulatif (cf. tableau 7.3) des préférences exprimées par les utilisateurs lors du questionnaire de fin de test à propos du besoin d'informations à chacune des étapes de la composition. Une majorité des utilisateurs (78% - 7/9, cf. tableau 7.3) ont demandé à ce que le modèle de tâches soit visuellement intégré au prototype lors de la phase de sélection. Cette intégration permettrait de mettre en correspondance la sélection des tâches et des éléments graphiques associés ou réciproquement. L'arbre de tâches semble être le modèle le plus intuitif aux utilisateurs pour analyser le comportement de l'application surtout si cela permet d'identifier les éléments graphiques leurs étant associés. Nous avons également noté que 67% (6/9) d'utilisateurs (cf. tableau 7.1) ayant utilisé l'extension par les tâches ont effectué une déviation "conceptuelle" du modèle de tâches vers le modèle de composants, déduisant des liens entre composants exclusivement à partir des informations sur les tâches.
D'ailleurs, les préférences exprimées montrent que pour l'étape de substitution, 67% des utilisateurs souhaiteraient avoir accès à l'assemblage de composants des applications.

A l'inverse nous remarquons qu'aucune information supplémentaire n'a été nécessaire pour l'utilisation des extensions s'appuyant sur la mise en page. Cependant 44% (4/9) des utilisateurs (pas forcément les mêmes que ceux qui ont utilisé cette extension) ont exprimé le fait que le modèle hiérarchique de l'interface graphique semble nécessaire avec des interfaces graphiques plus fournies. D'après les utilisateurs, ce modèle mettrait plus en évidence les informations sur l'imbrication des éléments des interfaces graphiques des applications.
Enfin, pour la phase de placement, les utilisateurs ne souhaitent pas avoir les informations supplémentaires sous la forme proposée lors des tests. Cependant ils souhaiteraient que les informations utilisées pendant la sélection (par exemple la structure hiérarchique avec l'extension par l'élément englobant) soient conservées pour le placement des éléments graphiques de la nouvelle application.

| **Etape de composition** / **Type d'Informations** | **Sélection** | **Substitution** | **Placement** |
|---|---|---|---|
| **Tâches** | Besoin d'informations sur les tâches pour 78% (7/9) des utilisateurs | Besoin d'informations sur les tâches pour 22% (2/9) des utilisateurs | Aucun besoin d'information |
| **Eléments logiciels** | Besoin d'informations sur les éléments logiciels pour 44% (4/9) des utilisateurs | Besoin d'informations sur les éléments logiciels pour 67% (6/9) des utilisateurs | Aucun besoin d'information |
| **Mise en page** | Oui, pour 44% (4/9) si application plus complexe | Non, pour aucun | Pas besoin d'informations supplémentaires mais par contre, pour tous les utilisateurs, besoin de conserver les informations de mise en page lors de l'étape de placement |

TABLE 7.3 – Préférences exprimées par les utilisateurs sur le besoin d'informations à chacune des étapes du processus de composition.

Nous en concluons que le modèle de tâches, décrivant ce que fait l'application, permet d'effectuer des sélections avec plus de justesse. Une fois la sélection effectuée, il faut alors manipuler l'application et nous constatons que pour les informaticiens, le modèle opérationnel est le plus pertinent. Pour le placement, l'affichage des informations n'est plus nécessaire mais celles-ci doivent être utilisées pour améliorer le placement.

### 7.9.4 Concernant le prototype

Au niveau de l'étape de sélection, un des utilisateurs a demandé à pouvoir visualiser la liste des éléments sélectionnés au fur et à mesure de la sélection indépendamment des interfaces graphiques.

Tous les utilisateurs ont exprimé le besoin de pouvoir essayer les extensions. A partir d'un élément

sélectionné, ils auraient voulu pouvoir tester les extensions et visualiser l'expansion (ou le repli) de la sélection au fur et à mesure. En particulier, l'intérêt de pouvoir sélectionner la profondeur d'application d'une extension n'a pas été comprise. Les utilisateurs en ont estimé le potentiel pour des applications comprenant un grand nombre d'éléments (logiciels et/ou graphiques et/ou tâches).

Au niveau de l'étape de substitutions, tous les utilisateurs ont eu des difficultés à comprendre les notions de "Keep Element" et "Remove Element" pour indiquer quel élément graphique est conservé et quel élément graphique est remplacé lors de la substitution.

Lors de la présentation du code généré pour une substitution entre deux éléments graphiques, il s'est avéré que sans explication supplémentaire, le code n'était pas compréhensible. L'ajout d'un schéma illustrant la partie impactée de l'assemblage de composant a toujours permis de faire comprendre à quoi l'adapteur servait et surtout comment compléter le code de celui-ci.

Au niveau de l'étape de placement, tous les utilisateurs ont demandé à avoir plus de possibilités d'actions, comme pouvoir redimensionner un élément graphique, pouvoir bénéficier d'un système de grille de placement ou d'un système de mise en page (layouts) prédéfinie etc...

Enfin, tout au long du processus, tous les utilisateurs auraient voulu pouvoir faire fonctionner les applications à composer. Effectivement, l'outil supprime toutes interactions avec l'application de telle sorte que seule l'interface graphique de l'application est présentée sans interaction possible.

### 7.9.5 Retour sur les difficultés du scénario

Un bilan spécifique sur les difficultés prévues du scénario peut être effectué.

D'une part, la difficulté sur la forte ressemblance des deux boutons a mis dans l'embarras plusieurs utilisateurs. Une différence existe clairement entre ce que nous avions envisagé comme manipulations (cf. annexe D page 181) avant de lancer les tests et les manipulations que les utilisateurs ont effectivement effectuées. Cette difficulté, pour 44% (4/9) d'entre eux, a conduit les utilisateurs à fusionner les deux boutons directement alors même que cette substitution n'était pas nécessaire pour le scénario de composition.

La seconde difficulté résidait dans le fait de devoir substituer un élément qui est une entrée texte par un élément qui est une liste. Cette substitution a causée quelques difficultés puisque la plupart des utilisateurs auraient voulu substituer non pas la liste avec l'entrée texte mais plutôt une information particulière présente dans la liste (entre autre, l'adresse du cinéma sélectionné) avec l'entrée texte. Or cette action n'est pas permise par l'approche puisque chaque élément graphique des interfaces des applications est considéré comme un tout insécable. La substitution attendue a toujours été réalisée, mais 44% (4/9) des participants ont fortement hésité et l'ont réalisée après avoir éliminé les autres possibilités.

Enfin la dernière difficulté venait du code de l'adapteur généré. La plupart des utilisateurs ont décidé de supprimer la méthode redondante mais parfois sans réellement comprendre pourquoi.

## 7.10 Bilan sur les tests utilisateurs

Nous analysons les résultats de notre expérimentation par rapport aux objectifs fixés pour le test.

Par rapport aux cinq objectifs que nous nous étions fixés, nous dressons le bilan des tests :

- **Compréhension du processus** : nous pouvons dire que le processus de composition en trois étapes (Sélection - Substitution - Placement) a bien été compris de tous les utilisateurs comme le fait ressortir notre questionnaire de fin d'entretien (cf. annexe G page 189).
- **Aptitude à réaliser le processus** : les utilisateurs ont tous réussi à manipuler l'outil de composition proposé et atteindre les objectifs qu'ils s'étaient fixés. Ceci a notamment été vérifié en leur demandant régulièrement de commenter leurs actions et en exprimant ce qu'ils voulaient faire et espérer obtenir. En terme de ressenti et de manipulations, les utilisateurs ont donc pu clairement mettre en œuvre ce qu'ils espéraient faire.
- **Informations fournies** : le résultat est ici mitigé. D'un côté les utilisateurs ont réussi à manipuler l'outil et 55% (5/9) ont réussi à faire la composition sans erreur. Cependant il ressort des entretiens qu'un apport d'information permettrait de rendre la composition plus facile pour prévoir le résultat des extensions et pour réaliser les substitutions. Il s'agit ici de fournir des informations sur les autres modèles afin de guider l'utilisateur, de le rassurer mais aussi afin de limiter sa charge cognitive.
- **Compréhension et pertinence des extensions** : plus de la moitié des utilisateurs (67% - 6/9 - d'entre eux) ont eu des difficultés à anticiper le fonctionnement des extensions. Cependant leur utilité a été comprise.
- **Apport des tâches** : le modèle de tâches semble être le modèle le plus intuitif pour effectuer la composition. Le modèle opérationnel semble être celui qui convient pour comprendre comment faire les liens entre les applications. Il nous semble que la complémentarité des informations que nous avons intégrées à notre modèle d'application est ici vérifiée.

## 7.11 Conclusion

Les résultats de cette première expérimentation sont encourageants. D'une part les participants ont bien accueillis notre modèle d'application et le processus composition. Ils sont généralement parvenus à réaliser l'application voulue. Les retours collectés proposent des pistes intéressantes pour améliorer l'utilisation des modèles et des outils. Mais les comportements vis à vis des difficultés identifiées nous indiquent qu'il est préférable de faire une composition dirigée non pas par un des trois points de vue (opérationnel, tâche, graphique), mais par les trois. En effet, nos travaux mettent en évidence que :

- les points de vue graphique et tâche sont les plus adaptés pour la sélection, ce que nous expliquons par leur proximité avec l'utilisateur final,
- le point de vue opérationnel et dans une moindre mesure celui des tâches sont les plus adaptés pour la composition, ce que nous expliquons par leur proximité avec un découpage fonctionnel de plus haut niveau (les tâches) ou bas niveau (les éléments logiciels) et
- la conservation de la mise en page lors de la sélection (surtout par celle faite par une extension graphique) pour la phase de placement.

# Quatrième partie

# Conclusions, perspectives et bibliographie

CHAPITRE 8

# Conclusions

**Sommaire**



NOUS concluons cette thèse par un rappel de nos principales contributions et nous identifions plusieurs perspectives à nos travaux.

## 8.1 Résumé des contributions pour la composition d'applications

Cette thèse est une contribution au domaine de la composition d'applications. Elle utilise des notions largement acceptées par plusieurs autres domaines connexes comme la conception d'Interface Homme-Machine et l'Ingénierie Logicielle. Nous avons étudié la **composition d'applications dirigée par la composition de leurs Interfaces Graphiques** (cf. partie II). Les contributions portent sur la définition d'un modèle d'application permettant la composition selon plusieurs points de vue de l'application (fonctionnalités, interface graphique et besoins). Des algorithmes de sélection de morceaux d'applications évoluant sur les trois points de vue de l'application, une composition d'applications basée sur un opérateur de substitution, et une mise en œuvre de ces travaux à travers un prototype complètent les apports de cette thèse. Ceux-ci ont été évalués par des tests utilisateurs.

### Un modèle formel pour la composition d'applications dirigée par la composition des interfaces graphiques

La première contribution (cf. chapitre 3) de cette thèse est de proposer un modèle d'application qui permet la composition d'applications. Nous basons notre composition sur des éléments logiciels qui forment un découpage de l'application. Pour chaque élément logiciel, les ports requis et les ports fournis sont connus. Nous n'avons aucune hypothèse supplémentaire sur les éléments logiciels, en particulier nous ne supposons pas avoir accès au code source, mais juste aux définitions des ports requis et des ports fournis. Cependant certaines informations présentes de notre modèle doivent être explicitées et disponibles.

Nous modélisons une application selon trois points de vue utilisés généralement séparément dans la composition d'applications (cf. chapitre 2) :

- un modèle opérationnel qui décrit les éléments logiciels et leurs inter-connexions. Ce modèle est inspiré de l'approche à composants. Il permet de différencier les ports liés à l'interface graphique (annotés "UI" ou "UI Component") des autres,
- un modèle de l'interface graphique qui décrit la structure hiérarchique "conteneur-contenu" des éléments graphiques ainsi que leurs relations géométriques et
- un modèle des besoins utilisateurs à travers un arbre de tâches dont les relations entre tâches sont celles de CTT [34].

Notre contribution principale porte sur la connexion explicite entre ces modèles :
- les liens entre le modèle opérationnel et le modèle de l'interface graphique : savoir quel élément logiciel englobe quel(s) élément(s) graphique(s) et réciproquement
- les liens entre le modèle opérationnel et l'arbre de tâches : savoir quels éléments logiciels sont utilisés pour atteindre un objectif (réaliser une tâche) et savoir comment est utilisé un élément logiciel (par quelles tâches)
- les liens entre le modèle de l'interface graphique et l'arbre de tâches : savoir quels éléments graphiques sont utilisés pour atteindre un objectif (réaliser une tâche) et savoir comment est utilisé un élément graphique (par quelles tâches).

### Des algorithmes de sélection sur plusieurs points de vue

La seconde contribution (cf. chapitre 4) de cette thèse est la définition et la formalisation de différentes fonctions de sélection afin d'aider le développeur dans les choix de parties d'applications à réutiliser. Nous avons défini les algorithmes de la complétion et la consolidation des sélections. Nous exploitons ainsi les liens pour passer d'une sélection d'éléments d'un point de vue à une sélection d'éléments des trois points de vue.

Nous avons également défini plusieurs explorations exploitant les différents points de vue : relation hiérarchique (tâche ou graphique), relation géométrique (graphique), relation temporelle (tâche) et relation fonctionnelle pour les éléments logiciels. Ces explorations deviennent des extensions combinables qui doivent être complétées et consolidées pour obtenir une sous-partie opérationnelle.

### Une composition d'applications par substitutions

La troisième contribution (cf. chapitre 5) de cette thèse est la proposition d'une composition basée sur un opérateur de substitutions. Disposant de différents morceaux d'applications à conserver ou à fusionner par substitution pour constituer la nouvelle application, l'idée est d'établir la glu entre ces différents morceaux pour éviter la création d'une application fruit d'une juxtaposition de différentes parties d'applications existantes sans lien entre elles.

La substitution se fait au niveau des éléments logiciels qui sont effectivement exécutés. Les substitutions sont guidées par les types des ports concernés et mises en œuvre par insertion d'adapteurs. Ces adapteurs peuvent être complétés une fois la substitution terminée.

### OntoCompo : un processus et un prototype pour la composition

Nous proposons un processus de composition séquentiel : sélection-substitution-placement. Ce processus est mis en œuvre dans un prototype appelé OntoCompo.

La piste exploitée dans cette thèse est que la partie la plus concrète et la plus directement manipulable d'une application est sa partie visible c'est-à-dire son Interface Graphique. C'est donc par la manipulation de cette interface graphique, pour la sélection puis pour effectuer les substitutions, que nous laissons le développeur dirigé la composition. C'est ainsi que les éléments d'interfaces graphiques redondants (car provenant de plusieurs applications) pourront être substitués afin de n'en garder qu'un seul ou de les remplacer par un autre.

Ce prototype propose une première implémentation des modèles et algorithmes présentés dans ce manuscrit. Toutes les extensions n'ont pas été traitées lors du développement du prototype. Effec-

tivement, le développement de ce prototype est centré autour du choix du processus de composition effectué *Sélection - Substitutions - Placement* afin de mettre en place des expérimentations utilisateurs nous permettant d'obtenir des informations sur les besoins des développeurs lors de leurs démarches de compositions d'applications. Ce développement a permis de mettre en œuvre les modèles des différents points de vue d'une application à travers la mise en place de différentes ontologies et l'utilisation de composants fractal comme base de développement et de manipulation des applications. Les substitutions s'appuient notamment sur l'introspection possible de ces composants. Les requêtes SPARQL effectuées tout au long du processus suivent les algorithmes que nous avons définis dans ce manuscrit. Ainsi, l'arbre de tâches de l'application sert de base descriptive de l'application mais n'est en aucun cas manipulé. L'application résultante de la composition est alors opérationnelle mais ne peut pas être directement composer de nouveau avec une autre application, une intervention du développeur étant nécessaire afin de re-construire un arbre de tâches cohérent pour l'application résultante.

### Retours d'expérimentations

A partir du prototype réalisé, nous avons mené une série d'expérimentations avec des informaticiens-développeurs. Les résultats de celle-ci sont encourageants. D'une part les participants ont bien accueillis notre modèle d'application et le processus composition. Ils sont généralement parvenus à réaliser l'application voulue. Les retours collectés proposent des pistes intéressantes pour améliorer l'utilisation des modèles et des outils.

## 8.2 Perspectives

### Obtention des modèles d'une application

Notre modèle d'application repose sur les informations contenues dans les modèles opérationnels, des tâches et des interfaces graphiques. Une étude sur les moyens d'obtenir ces modèles de manière automatique ou quasi-automatique (par reverse engineering par exemple) est à effectuer.

Effectivement, le modèle opérationnel que nous avons retenu s'inspire fortement des modèles à composant non hiérarchique. Nous n'explicitons que les informations nécessaires pour la composition mais ces informations sont parfois non présentes dans les modèles à composant. En revanche, les modèles à composants ont souvent une description de l'assemblage qui constituent une application. Cette description pourrait servir de base à l'instanciation de notre modèle.

Ce qui nous intéresse dans la modélisation de l'interface graphique d'une application est sa "mise en page" (*layout*). Cette modélisation de la "mise en page" a été construite à partir de l'étude de certains langages de programmation et la manière de construire des interfaces graphiques avec ces langages comme Java (Swing) [72], MXML pour Flex [70], XAML pour WPF [68] ... Notre modélisation est à priori extractible par introspection et analyse des interfaces graphiques existantes.

Enfin, les arbres de tâches sont le plus souvent descriptifs et utilisés lors de la conception des applications. Ils sont parfois présents lors de l'exécution des applications. Dans tous les cas, il faudra transformer, si possible automatiquement sinon manuellement, ces modèles existants en notre représentation.

### Manipulation simultanée des trois points de vue

En conclusion des expérimentations, les comportements des utilisateurs vis à vis des difficultés identifiées nous indiquent qu'il est préférable de faire une composition dirigée non pas par un des trois points de vue (opérationnel, tâche, graphique), mais par les trois. En effet, nos travaux mettent en évidence que :

- les points de vue graphique et tâche sont les plus adaptés pour la sélection, ce que nous expliquons par leur proximité avec l'utilisateur final,
- le point de vue opérationnel et dans une moindre mesure celui des tâches sont les plus adaptés pour la composition, ce que nous expliquons par leur proximité avec un découpage fonctionnel de plus haut niveau (les tâches) ou bas niveau (les éléments logiciels) et
- la conservation de la mise en page lors de la sélection (surtout par celle faite par une extension graphique) pour la phase de placement.

Un travail de développement est nécessaire pour adapter notre prototype OntoCompo à cette nouvelle hypothèse. Nous notons que notre modèle permet cette nouvelle approche de la composition et que cette hypothèse nous conforte dans notre volonté d'utiliser toutes les dimensions d'une application pour la composition. Il s'agit donc de fournir des vues et manipulations "synchronisées" sur différents points de vue de l'application, tout au long du processus de composition. Le développement de la nouvelle version d'OntoCompo visera la réalisation de nouvelles expérimentations pour vérifier cette exploitation des différents modèles.

### Construction de l'arbre de tâches de l'application composée

Une des limitations de notre approche est que nous ne savons pas reconstruire un arbre de tâches complet pour la nouvelle application. Cette reconstruction est un objectif intéressant à atteindre afin de pouvoir directement recomposer l'application résultat d'une composition. Ainsi le cycle de vie des applications en serait prolongé.

Notre intuition est que, en fonction des substitutions réalisées, il doit être possible de déduire une opération sur les parties concernées des arbres de tâches. Le risque est alors d'obtenir des fragments d'arbre de tâches qui ne sont pas reliés entre-eux. Il convient donc d'étudier l'évolution des arbres de tâches, de déterminer ce qui peut être déduit des opérations de composition (sélection, substitution et placement) et ce qui ne peut pas l'être. Dans ce cas, il faudra avoir recours à une intervention du meneur de la composition afin d'obtenir un arbre de tâches complet, cohérent et lié aux autre modèles de l'application.

### Composition menée par l'utilisateur final

Nous avons été surpris lors des expérimentations par le comportement des participants expérimentés en composition. Ceux-ci demandaient généralement plus d'informations complémentaires et faisaient moins confiance dans OntoCompo que les autres. Les informations demandées (par les uns et par les autres) au début de la composition (sélection) sont plutôt de "haut niveau" (tâches). Mais pour la réalisation de la composition, une manipulation "bas niveau" (au niveau des composants) est demandée. Si nous projetons ces résultats à des utilisateurs "non informaticiens", nous sommes amenés à penser qu'ils pourraient plus facilement faire confiance au système de composition.

Viser de tels utilisateurs permettrait d'utiliser nos résultats dans le cadre de l'informatique ubiquitaire, où les situations de mobilité (géographique et d'activité) renforcent les besoins d'adaptation

dynamique, la composition étant une forme possible d'adaptation. La composition nécessiterait d'être renforcé en terme d'informations fournies à l'utilisateur.

Cependant, en cas de besoin d'informations pour la réalisation des compositions, les informations fournies par le modèle opérationnel ne sauront pas directement compréhensibles pour un utilisateur "non informaticien". En conséquence, pour atteindre des utilisateurs "non informaticiens", il nous faut (i) faire en sorte qu'ils n'aient pas besoin d'informations (ou le moins possible), donc automatiser un maximum d'opérations et (ii) de fournir, si nécessaire, des explications adaptées à son niveau de connaissance. Sur ce dernier point, utiliser des modèles auto-explicatifs, à l'instar des interfaces auto-explicatives de [69], est la solution que nous aimerions suivre.

CHAPITRE 9

# Bibliographie détaillée par catégorie

## 9.1 Bibliographie : Représentation du Noyau Fonctionnel d'une application


[1] Leonard J. Bass and Joëlle Coutaz. A metamodel for the runtime architecture of an interactive system : the uims tool developers workshop. *SIGCHI Bull.*, 24(1) :32–37, January 1992.

[2] M. Beisiegel and al. Service component architecture - building systems using a service oriented architecture. *A Joint Whitepaper by BEA, IBM, Interface21, IONA, SAP, Siebel, Sybase*, Nov 2005.

[3] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java : Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12) :1257–1284, September 2006.

[4] Joëlle Coutaz. Pac : An object oriented model for implementing user interfaces. *SIGCHI Bull.*, 19(2) :37–41, October 1987.

[5] Thomas Erl. *Service-Oriented Architecture : Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[6] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, Irvine, California, 2000.

[7] George T. Heineman and William T. Councill. *Component-based software engineering : putting the pieces together*. Addison-Wesley Professional, 2001.

[8] Vincent Hourdin, Jean-Yves Tigli, Stéphane Lavirotte, Gaëtan Rey, and Michel Riveill. Slca, composite services for ubiquitous computing. In *Proceedings of the International Conference on Mobile Technology, Applications, and Systems*, Mobility '08, pages 11 :1–11 :8, New York, NY, USA, 2008. ACM.

[9] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990.

[10] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming, in proceedings of the european conference on object-oriented programming (ecoop'97), finland. *Springer-Verlag LNCS*, 1241 :16, 1997.

[11] Kung-Kiu Lau and Zheng Wang. Software component models. *IEEE Trans. Software Eng.*, 33(10) :709–724, 2007.

[12] Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter Brown, and Rebekah Metz. Reference model for service oriented architecture 1.0. Technical Report wd-soa-rm-cd1, OASIS, February 2006.

[13] Oasis. The Universal Description, Discovery and Integration (UDDI) protocol. https://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm, 2004.

[14] Open SOA. *SCA Service Component Architecture - Assembly Model Specification*, March 2007. Version 1.00.

[15] Trygve M. H. Reenskaug. MVC XEROX PARC 1978–1979. http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html, 1979.

[16] Clemens and Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[17] W3C. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, 2003.

[18] W3C. OWL-S : Semantic Markup for Web Services. http://www.w3.org/Submission/OWL-S/, 2004.

[19] W3C. SOAP Version 1.2 Part 1 : Messaging Framework (Second Edition). http://www.w3.org/TR/soap12-part1/, 2007.

## 9.2 Bibliographie : Représentation de l'Interface Homme-Machine d'une application

[20] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. Uiml : an appliance-independent xml user interface language. In *Proceedings of the eighth international conference on World Wide Web*, WWW '99, pages 1695–1708, New York, NY, USA, 1999. Elsevier North-Holland, Inc.

[21] John Annett. *Hierarchical Task Analysis*, chapter 2, pages 17–35. Handbook of cognitive task design. CRC Press, June 2003.

[22] John Annett and Keith D. Duncan. Task analysis and training design. *Occupational Psychology*, 41 :211–221, 1967.

[23] Grégory Bourguin, Arnaud Lewandowski, and Jean-Claude Tarby. Defining task oriented components. In *Task Models and Diagrams for User Interface Design*, pages 170–183. Springer, 2007.

[24] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting With Computers Vol. 15/3*, 15(3) :289–308, June 2003.

[25] Stuart K. Card, Thomas P. Moran, and Newell Allen. *The Psychology of Human-Computer Interaction*. Lawrence. Erlbaum Associates, 1983.

[26] Alfonso García Frey, Eric Céret, Sophie Dupuy-Chessa, Gaëlle Calvary, and Yoann Gabillon. Usicomp : an extensible model-driven composer. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*, pages 263–268. ACM, 2012.

[27] H. Rex Hartson and Philip D. Gray. Temporal aspects of tasks in the user action notation. *Hum.-Comput. Interact.*, 7(1) :1–45, March 1992.

[28] ISO. Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour. http://www.iso.org/iso/catalogue_detail.htm?csnumber=16258, 1988.

[29] Bonnie E. John and David E. Kieras. The goms family of analysis techniques : Tools for design and evaluation. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[30] Quentin Limbourg, Costin Pribeanu, and Jean Vanderdonckt. Towards uniformed task models in a model-based approach. In *Proceedings of the 8th International Workshop on Interactive Systems : Design, Specification, and Verification-Revised Papers*, DSV-IS '01, pages 164–182, London, UK, UK, 2001. Springer-Verlag.

[31] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and ViÂctor Lopez-Jaquero. UsiXML : A Language Supporting Multi-path Development of User Interfaces. In Rémi Bastide, Philippe Palanque, and Jörg Roth, editors, *Engineering Human Computer Interaction and Interactive Systems*, volume 3425 of *Lecture Notes in Computer Science*, chapter 12, pages 134–135. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2005.

[32] Giulio Mori, Fabio Paterno, and Carmen Santoro. Design and development of multidevice user interfaces through multiple logical descriptions. *IEEE Trans. Softw. Eng.*, 30(8) :507–520, August 2004.

[33] Fabio Paternò. *ConcurTaskTrees : an engineered approach to model-based design of interactive systems*, pages 483–500. The Handbook of Analysis for Human-Computer Interaction. Lawrence Erlbaum Associates, 2002.

[34] Fabio Paternò, Cristiano Mancini, and Silvia Meniconi. Concurtasktrees : A diagrammatic notation for specifying task models. In *Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction*, INTERACT '97, pages 362–369, London, UK, UK, 1997. Chapman & Hall, Ltd.

[35] Fabio Paterno', Carmen Santoro, and Lucio Davide Spano. Maria : A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Trans. Comput.-Hum. Interact.*, 16(4) :19 :1–19 :30, November 2009.

[36] Anne-Marie Pinna-Déry and Jérémy Fierstone. Component model and programming : a first step to manage Human Computer Interaction Adaptation. In *5th International Symposium on Human-Computer Interaction with Mobile Devices and Services(Mobile HCI)*, volume LNCS 2795 of , pages 456–460, Udine, Italy, September 2003. L. Chittaro (Ed.), Springer Verlag.

[37] Vincent Tietz, Gregor Blichmann, Stefan Pietschmann, and Klaus Meissner. Task-based recommendation of mashup components. In *Proceedings of the 11th international conference on Current Trends in Web Engineering*, ICWE'11, pages 25–36, Berlin, Heidelberg, 2012. Springer-Verlag.

[38] Martijn Van Welie, Gerrit C. Van Der Veer, and Anton Eliëns. An ontology for task world models. In *Proceedings of DSV-IS98, Abingdon*, pages 3–5. UK, Springer-Verlag, 1998.

## 9.3 Bibliographie : Composition d'applications

[39] Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben, Francisco Curbera, Mark Ford, Yaron Goland, Alejandro Guízar, Neelakantan Kartha, Canyang K. Liu, Rania Khalaf, Dieter König, Mike Marin, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS Web Services Business Process Execution Language (WSBPEL) TC, April 2007.

[40] Andreas Auinger, Martin Ebner, Dietmar Nedbal, and Andreas Holzinger. Mixing content and endless collaboration — mashups : Towards future personal learning environments. In *Proceedings of the 5th International Conference on Universal Access in Human-Computer Interaction. Part III : Applications and Services*, UAHCI '09, pages 14–23, Berlin, Heidelberg, 2009. Springer-Verlag.

[41] ServFace Consortium. IServFace Research Project. `http://www.servface.eu`, 2008.

[42] Javier Criado, Nicolás Padilla, Luis Iribarne, and Jose-Andrés Asensio. User interface composition with cots-ui and trading approaches : Application for web-based environmental information systems. In *Knowledge Management, Information Systems, E-Learning, and Sustainability Research*, pages 259–266. Springer, 2010.

[43] Alexandre Demeure, Gaëlle Calvary, and Karin Coninx. Comet (s), a software architecture style and an interactors toolkit for plastic user interfaces. In *Interactive Systems. Design, Specification, and Verification*, pages 225–237. Springer, 2008.

[44] Marius Feldmann, Gerald Hubsch, Thomas Springer, and Alexander Schill. Improving task-driven software development approaches for creating service-based interactive applications by using annotated web services. In *Next Generation Web Services Practices, 2009. NWESP'09. Fifth International Conference on*, pages 94–97. IEEE, 2009.

[45] Marius Feldmann, Jordan Janeiro, Tobias Nestler, Gerald Hübsch, Uwe Jugel, André Preussner, and Alexander Schill. An integrated approach for creating service-based interactive applications. In *Proceedings of the 12th IFIP TC 13 International Conference on Human-Computer Interaction : Part II*, INTERACT '09, pages 896–899, Berlin, Heidelberg, 2009. Springer-Verlag.

[46] Yoann Gabillon, Matthieu Petit, Gaëlle Calvary, Humbert Fiorino, et al. Automated planning for user interface composition. In *Proceeding of the 2nd SEMAIS workshop of the IUI 2011 conference*, 2011.

[47] Jeronimo Ginzburg, Gustavo Rossi, Matias Urbieta, and Damiano Distante. Transparent interface composition in web applications. In *Proceedings of the 7th international conference on Web engineering*, pages 152–166. Springer-Verlag, 2007.

[48] Cédric Joffroy, Benjamin Caramel, Anne-Marie Dery-Pinna, and Michel Riveill. When the functional composition drives the user interfaces composition : process and formalization. In *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '11, pages 207–216, New York, NY, USA, 2011. ACM.

[49] Rania Khalaf, Nirmal Mukhi, and Sanjiva Weerawarana. Service-Oriented Composition in BPEL4WS. In *Proceedings of the 20th Internation World Wide Web Conference (Alternate Papers Track)*, WWW'03, Budapest, Hungary, May 2003.

[50] Sophie Lepreux, Anas Hariri, José Rouillard, Dimitri Tabary, Jean-Claude Tarby, and Christophe Kolski. Towards multimodal user interfaces composition based on usixml and mbd principles. In *Human-Computer Interaction. HCI Intelligent Multimodal Interaction Environments*, pages 134–143. Springer, 2007.

[51] Sophie Lepreux and Jean Vanderdonckt. Towards a support of user interface design by composition rules. In *Computer-Aided Design of User Interfaces V*, pages 231–244. Springer, 2007.

[52] Sophie Lepreux, Jean Vanderdonckt, and Christophe Kolski. User interface composition with usixml. *UsiXML 2010*, 2010.

[53] Arnaud Lewandowski, Sophie Lepreux, and Grégory Bourguin. Tasks models merging for high-level component composition. In *Human-Computer Interaction. Interaction Design and Usability*, pages 1129–1138. Springer, 2007.

[54] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 178–187, New York, NY, USA, 2000. ACM.

[55] Duane Merrill. Mashups : The new breed of Web app—An introduction to mashups. Technical report, IBM, August 2006.

[56] Tobias Nestler, Marius Feldmann, André Preußner, and Alexander Schill. Service composition at the presentation layer using web service annotations. In *Proceedings of the 1st Intl. Workshop on Lightweight Integration on the Web*, pages 63–68, San Sebastian, Spain, 2009.

[57] Audrey Occello, Cédric Joffroy, Anne-Marie Pinna-Déry, Philippe Renevier, and Michel Riveill. Experiments in Model Driven Composition of User Interfaces. In *Proceedings of the 10th IFIP International Conference on Distributed Applications and Interoperable Systems*, DAIS'10, pages 98–111, Amsterdam, Netherlands, June 2010.

[58] Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. Exploiting web service annotations in model-based user interface development. In *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, pages 219–224. ACM, 2010.

[59] Fabio Paternó, Carmen Santoro, Lucio Davide Spano, and HIIS CNR-ISTI. User task-based development of multi-device service-oriented applications. In *International Conference on Advanced Visual Interfaces. LNCS*, volume 5726, 2010.

[60] Stefan Pietschmann, Martin Voigt, Andreas Rümpel, and Klaus Meißner. Cruise : Composition of rich user interface services. In *Web Engineering*, pages 473–476. Springer, 2009.

[61] Shirin Sohrabi and Sheila A. McIlraith. Preference-based web service composition : A middle ground between execution and search. In *Proceedings of the 9th International Semantic Web Conference (ISWC-10)*, pages 713–729, Shanghai, China, November 2010.

[62] Wolfgang Stuerzlinger, Olivier Chapuis, Dusty Phillips, and Nicolas Roussel. User Interface Façades : Towards Fully Adaptable User Interfaces. In *UIST '06 : ACM Symposium on User Interface Software and Technology*, pages 309–318, Montreux, Suisse, October 2006. ACM - SIGCHI & SIGGRAPH, ACM.

[63] Desney S Tan, Brian Meyers, and Mary Czerwinski. Wincuts : manipulating arbitrary window regions for more effective use of screen space. In *CHI'04 extended abstracts on Human factors in computing systems*, pages 1525–1528. ACM, 2004.

[64] Qi Zhao, Gang Huang, Jiyu Huang, Xuanzhe Liu, and Hong Mei. A web-based mashup environment for on-the-fly service composition. In *Service-Oriented System Engineering, 2008. SOSE'08. IEEE International Symposium on*, pages 32–37. IEEE, 2008.

## 9.4 Bibliographie : Autre

[65] Sandrine Balbo. *Evaluation ergonomique des interfaces utilisateur : un pas vers l'automatisation*. PhD thesis, Université Joseph-Fourier-Grenoble I, 1994.

[66] Sandrine Balbo, Joëlle Coutaz, and Daniel Salber. Towards automatic evaluation of multimodal user interfaces. In *Proceedings of the 1st international conference on Intelligent user interfaces*, pages 201–208. ACM, 1993.

[67] Olivier Corby, Alban Gaignard, Catherine Faron-Zucker, Johan Montagnat, et al. Kgram versatile inference and query engine for the web of linked data. In *Proceedings of the International Conference on Web Intelligence*, pages 1–8, 2012.

[68] MSDN Microsoft Corporation. Xaml in wpf. `http://msdn.microsoft.com/en-us/library/ms747122(v=VS.100).aspx`.

[69] Alfonso García Frey, Gaëlle Calvary, and Sophie Dupuy-Chessa. Users need your models ! exploiting design models for explanations. In *Proceedings of HCI 2012, Human Computer Interaction, People and Computers XXVI, The 26th BCS HCI Group conference (Birmingham, UK)*, pages 79–88. British Computer Society Swinton, UK, UK ©2012, 2012.

[70] Adobe Systems Inc. Flex. `http://www.adobe.com/products/flex/`, 2006.

[71] Célia Martinie, Philippe Palanque, David Navarre, Marco Winckler, and Erwann Poupart. Model-based training : an approach supporting operability of critical interactive systems. In *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '11, pages 53–62, New York, NY, USA, 2011. ACM.

[72] Oracle. Laying out components within a container. `http://docs.oracle.com/javase/tutorial/uiswing/layout/index.html`.

## 9.5 Bibliographie personnelle : Publications liées à cette thèse

### Publications dans des Conférences Internationales

[BPDFZ+11] Christian Brel, Anne-Marie Pinna-Déry, Catherine Faron-Zucker, Philippe Renevier, and Michel Riveill. OntoCompo : An Ontology-Based Interactive System To Compose Applications. In *Seventh International Conference on Web Information Systems and Technologies (WEBIST 2011), short paper*, pages 322–327. Springer-Verlag, May 2011.

[BPDRR11] Christian Brel, Anne-Marie Pinna-Déry, Philippe Renevier, and Michel Riveill. OntoCompo : A Tool To Enhance Application Composition. In *13th IFIP TC13 Conference in Human-Computer Interaction INTERACT 2011 (Interact 2011)*, pages 588–591, September 2011.

[BRO+10] Christian Brel, Philippe Renevier, Audrey Occello, Anne-Marie Pinna-Déry, Catherine Faron-Zucker, and Michel Riveill. Application Composition Driven By UI Composition. In *3rd International Conference on Human Computer Software Engineering (HCSE 2010)*, volume 6409 of *LNCS*, pages 198–205. LNCS, October 2010.

[BRPDR12a] Christian Brel, Philippe Renevier, Anne-Marie Pinna-Déry, and Michel Riveill. Annotated Component-Based Description for Application Composition. In *The Seventh International Conference on Software Engineering Advances (ICSEA 2012)*, November 2012.

[BRPDR12b] Christian Brel, Philippe Renevier, Anne-Marie Pinna-Déry, and Michel Riveill. Application and UI composition using a Component-Based Description and Annotations. In *38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2012)*, pages 204–207, September 2012.

[BRPDR12c] Christian Brel, Philippe Renevier, Anne-Marie Pinna-Déry, and Michel Riveill. UI Modeling as Ontology for Composition. In *The Twenty First International Conference On Software Engineering and Data Engineering (SEDE 2012)*, pages 67–72, June 2012.

### Publications dans des Conférences Nationales

[BM11] Christian Brel and Sébastien Mosser. Vers une approche flot de données pour supporter la composition d'interfaces homme-machine. In *Journées sur l'Ingénierie Dirigée par les Modèles (IDM'11)*, pages 1–7, Lille, June 2011.

[BR11] Christian Brel and Philippe Renevier. Composing Applications with OntoCompo. In *Conference Interaction Homme-Machine (IHM)*, Nice, October 2011.

[Bre09] Christian Brel. Une approche de description d'Interfaces Homme-Machine multi-niveaux. In *MAnifestation des JEunes Chercheurs en Sciences et Technologies de l'Information et de la Communication 2009 (MajecSTIC 2009)*, Avignon, November 2009.

# Cinquième partie

# Annexes

ANNEXE A

# Descriptions complètes des applications "Cinema" et "Maps"

CETTE annexe présente les modèles et interfaces des 2 applications de notre étude de cas "Cinema" et "Maps".

## A.1 Description de l'application "Cinema"

La première application est une application de recherche de cinémas (cf. figure A.1). Elle permet de découvrir une liste de cinémas les plus proches d'une adresse donnée. La première liste, située à gauche de l'interface graphique, affiche un cinéma par ligne avec son nom, son adresse accompagné du code postal et de la ville associée, le nombre de salles disponibles. Lors de la sélection d'un cinéma dans la liste, une seconde liste, situé à droite sur l'interface graphique, donne la liste des séances de films prévues dans le cinéma sélectionné.

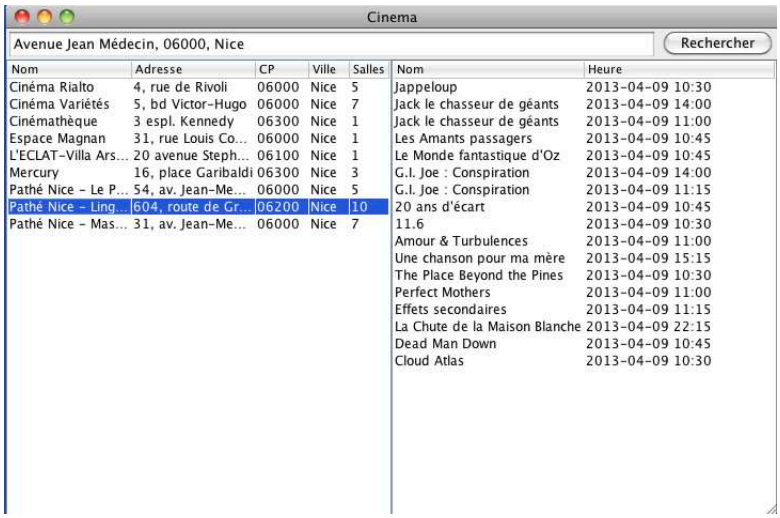



FIGURE A.1 – Interface graphique de l'application "Cinema".

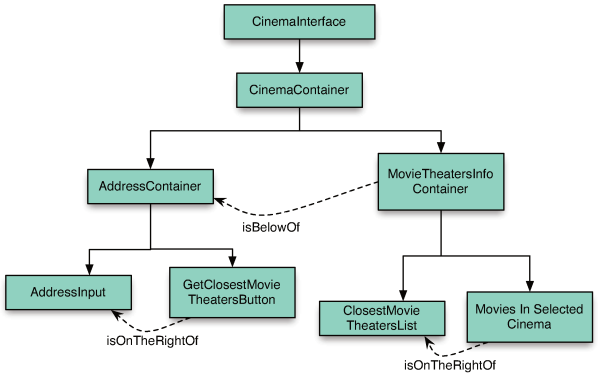



FIGURE A.2 – Modèle de l'interface graphique de l'application "Cinema".

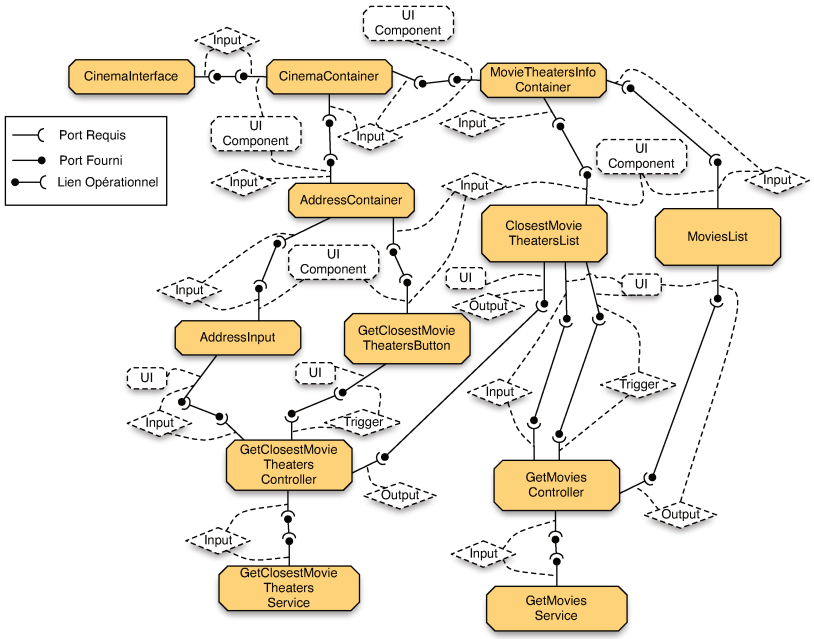



FIGURE A.3 – Modèle opérationnel de l'application "Cinema".

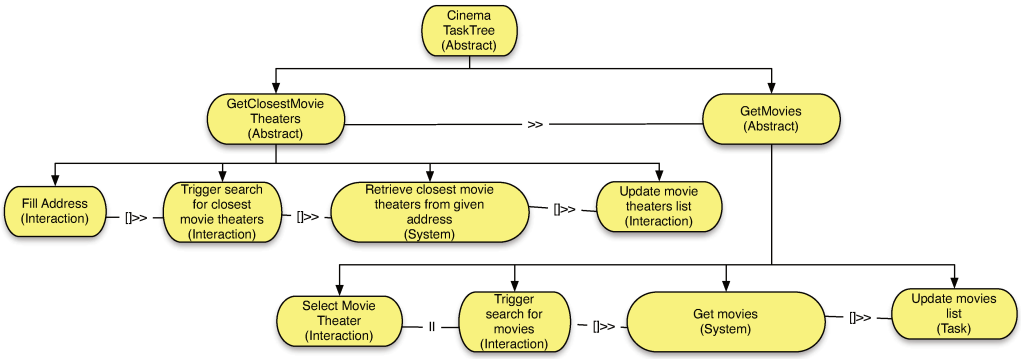



FIGURE A.4 – Modèle de l'arbre de tâches de l'application "Cinema".

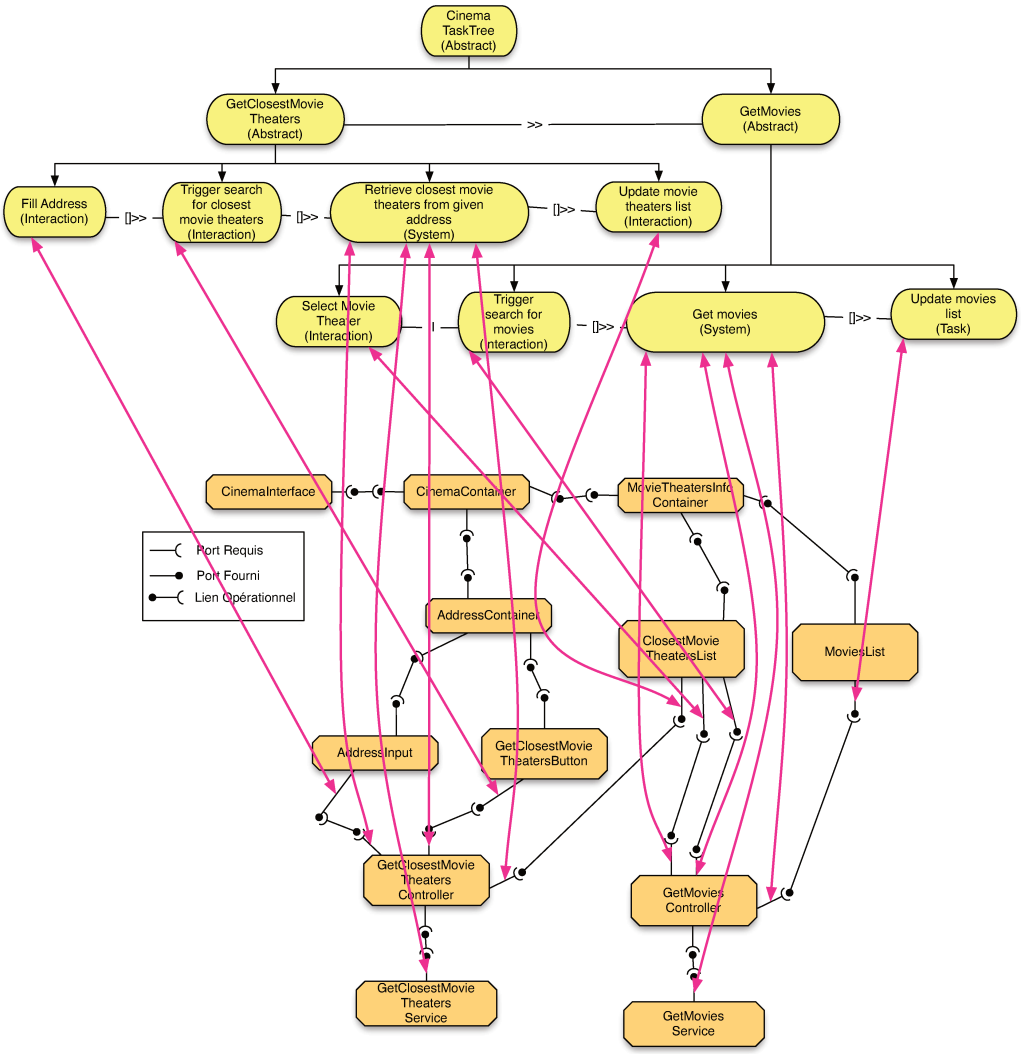



FIGURE A.5 – Liens entre modèle de tâches et modèle opérationnel de l'application "Cinema".

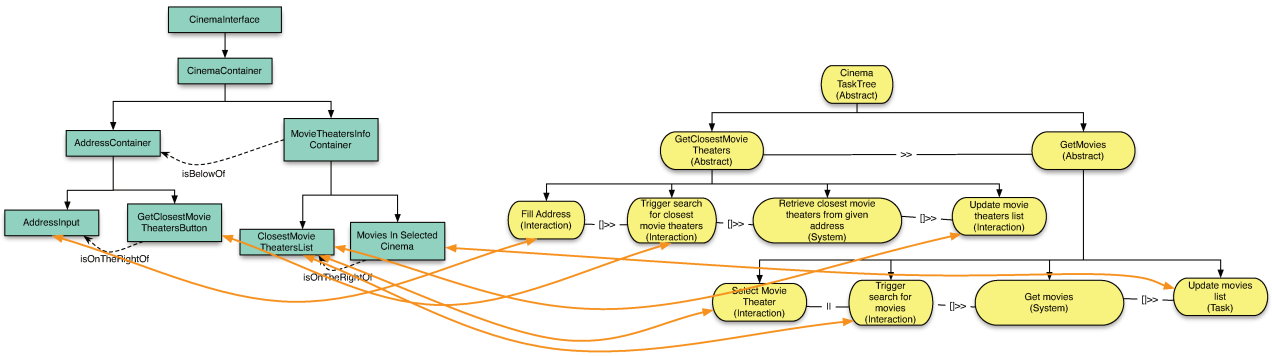



FIGURE A.6 – Liens entre modèle de tâches et modèle de l'interface graphique de l'application "Cinema".

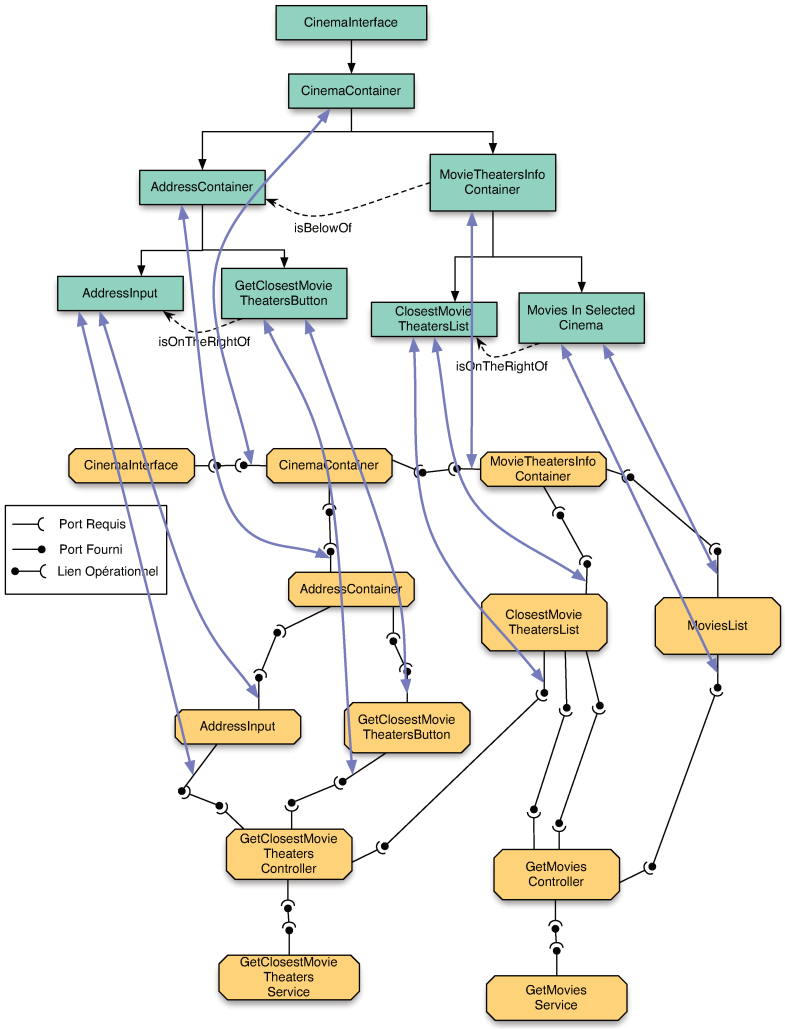



FIGURE A.7 – Liens entre modèle opérationnel et modèle de l'interface graphique de l'application "Cinema".

## A.2 Description de l'application "Maps"

La seconde application est une application de calcul d'itinéraire (cf. figure A.8). Elle permet à partir de deux adresses de calculer le trajet et de l'afficher sur une carte routière. Elle permet aussi d'afficher les principales intersections de l'itinéraire. Il est possible d'effectuer un zoom sur la carte à l'aide du "slider" présent dans l'interface graphique ou en agissant directement sur la carte.

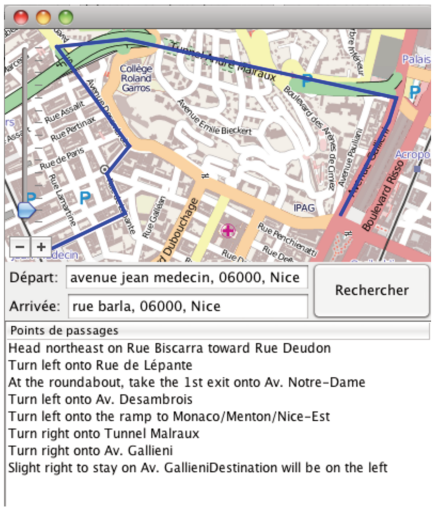



FIGURE A.8 – Interface graphique de l'application de calcul d'itinéraire "Maps".

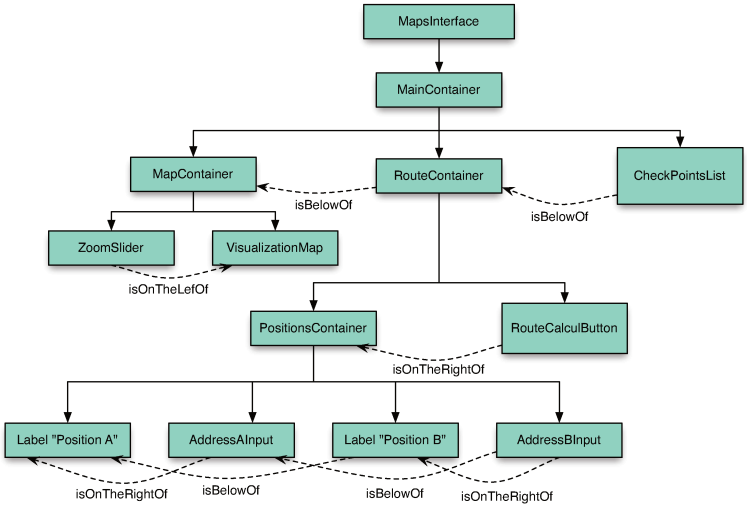



FIGURE A.9 – Modèle de l'interface graphique de l'application "Maps".

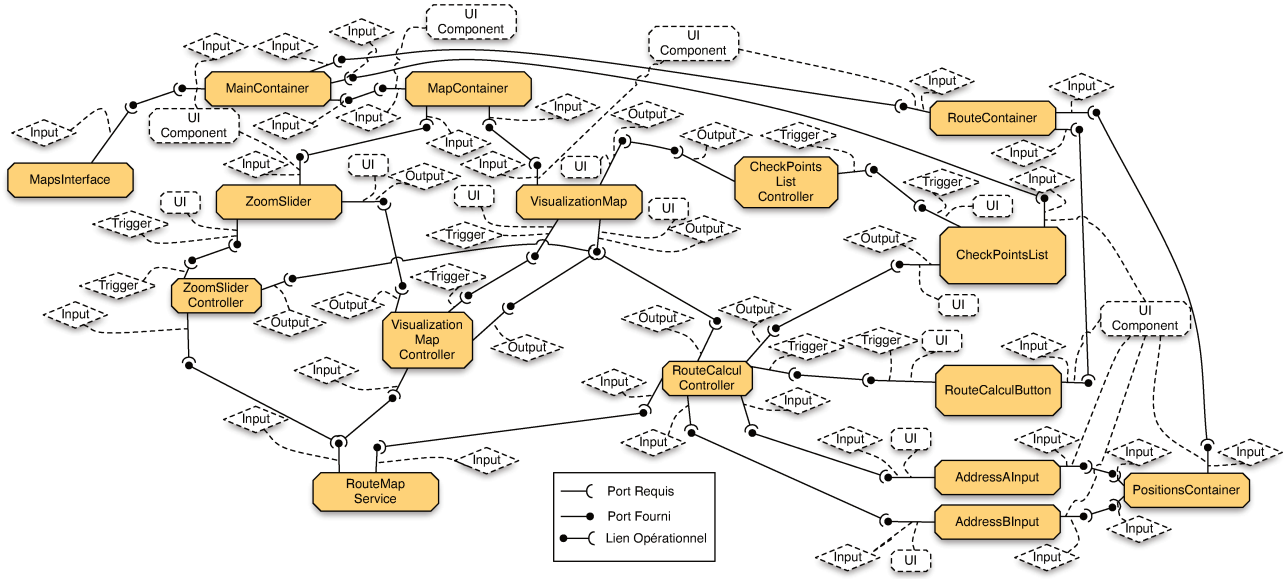



FIGURE A.10 – Modèle opérationnel de l'application "Maps".

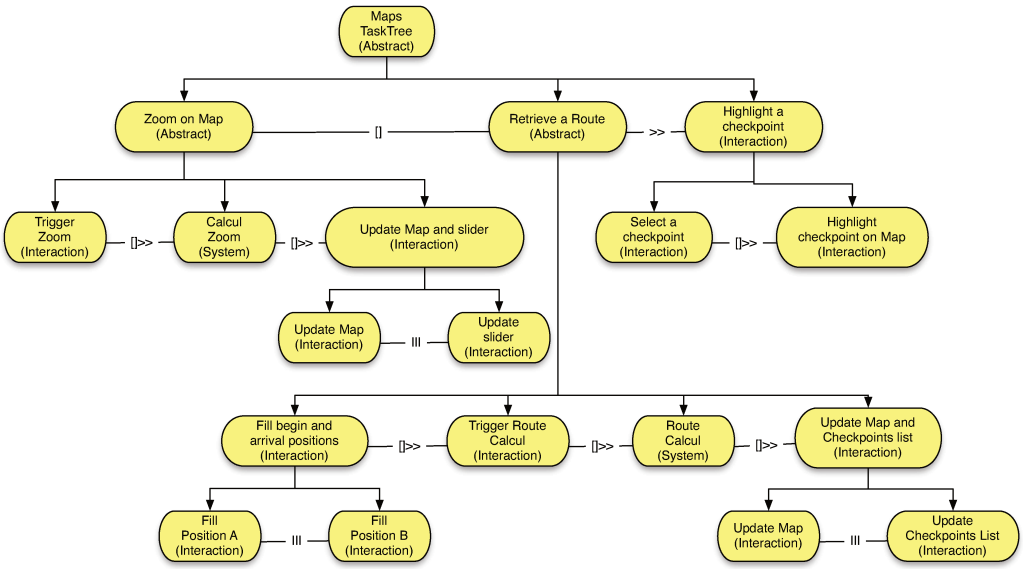



FIGURE A.11 – Modèle de l'arbre de tâches de l'application "Maps".

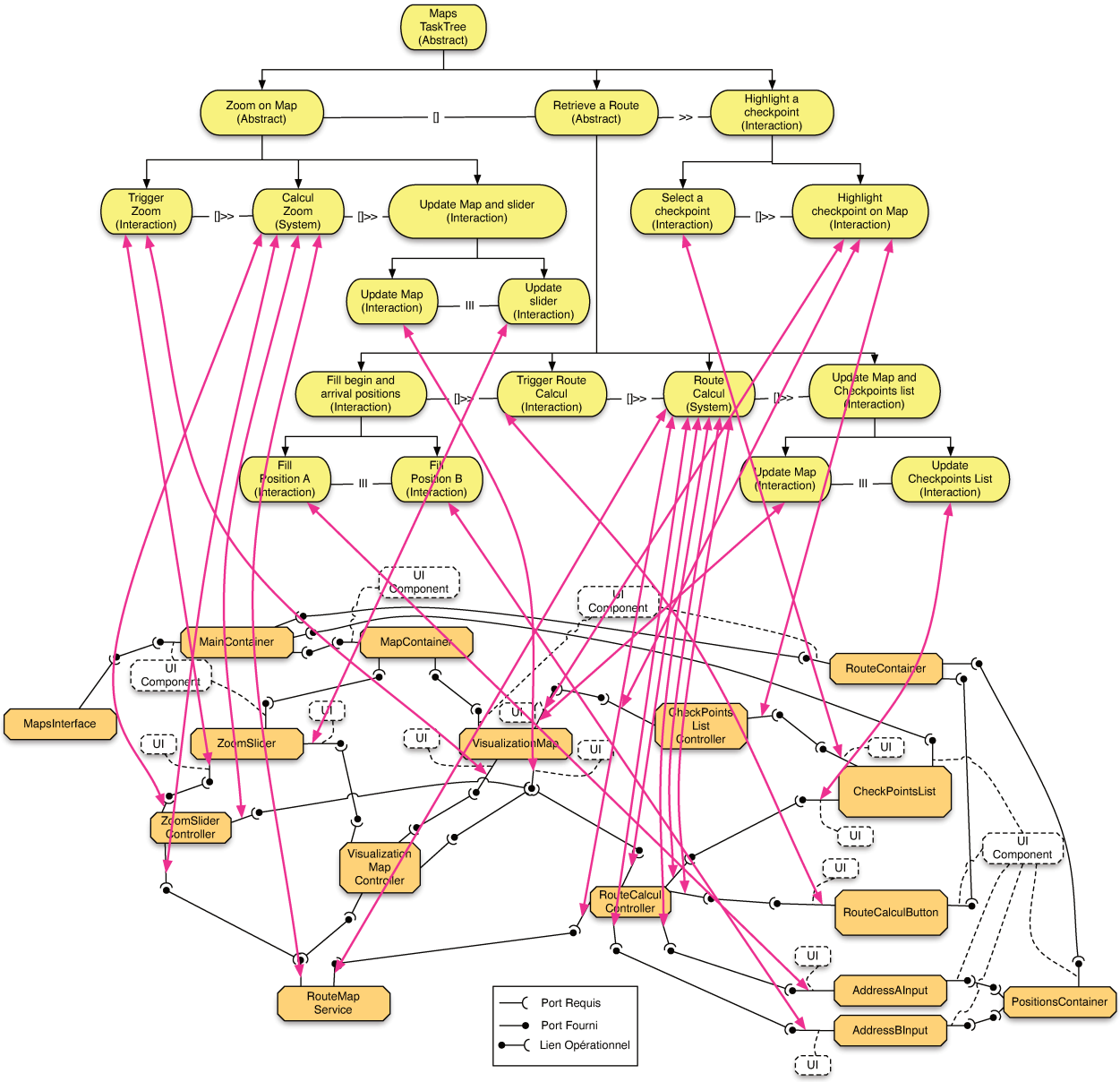



FIGURE A.12 – Liens entre modèle de tâches et modèle opérationnel de l'application "Maps".

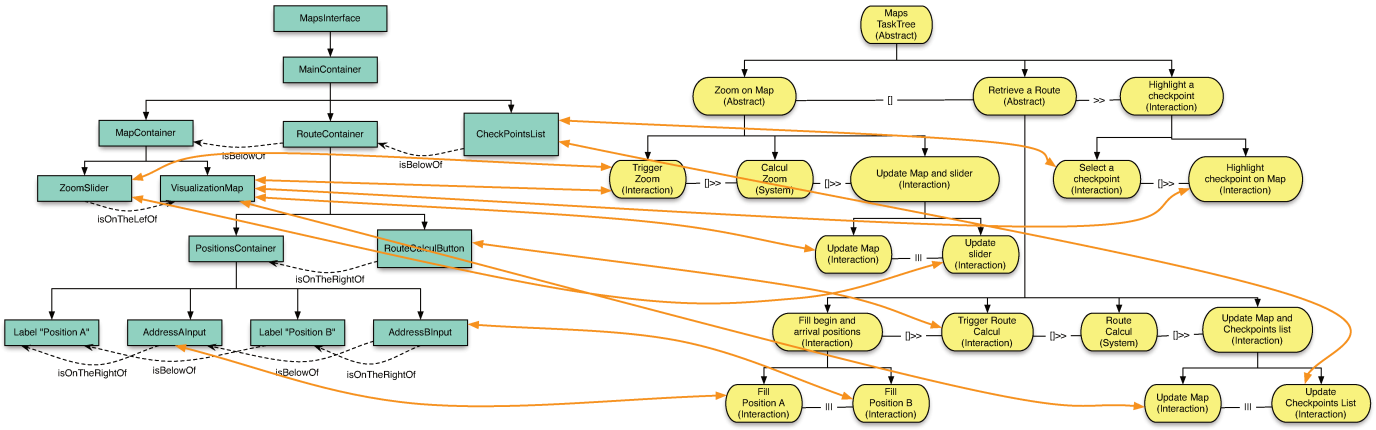



FIGURE A.13 – Liens entre modèle de tâches et modèle de l'interface graphique de l'application "Maps".

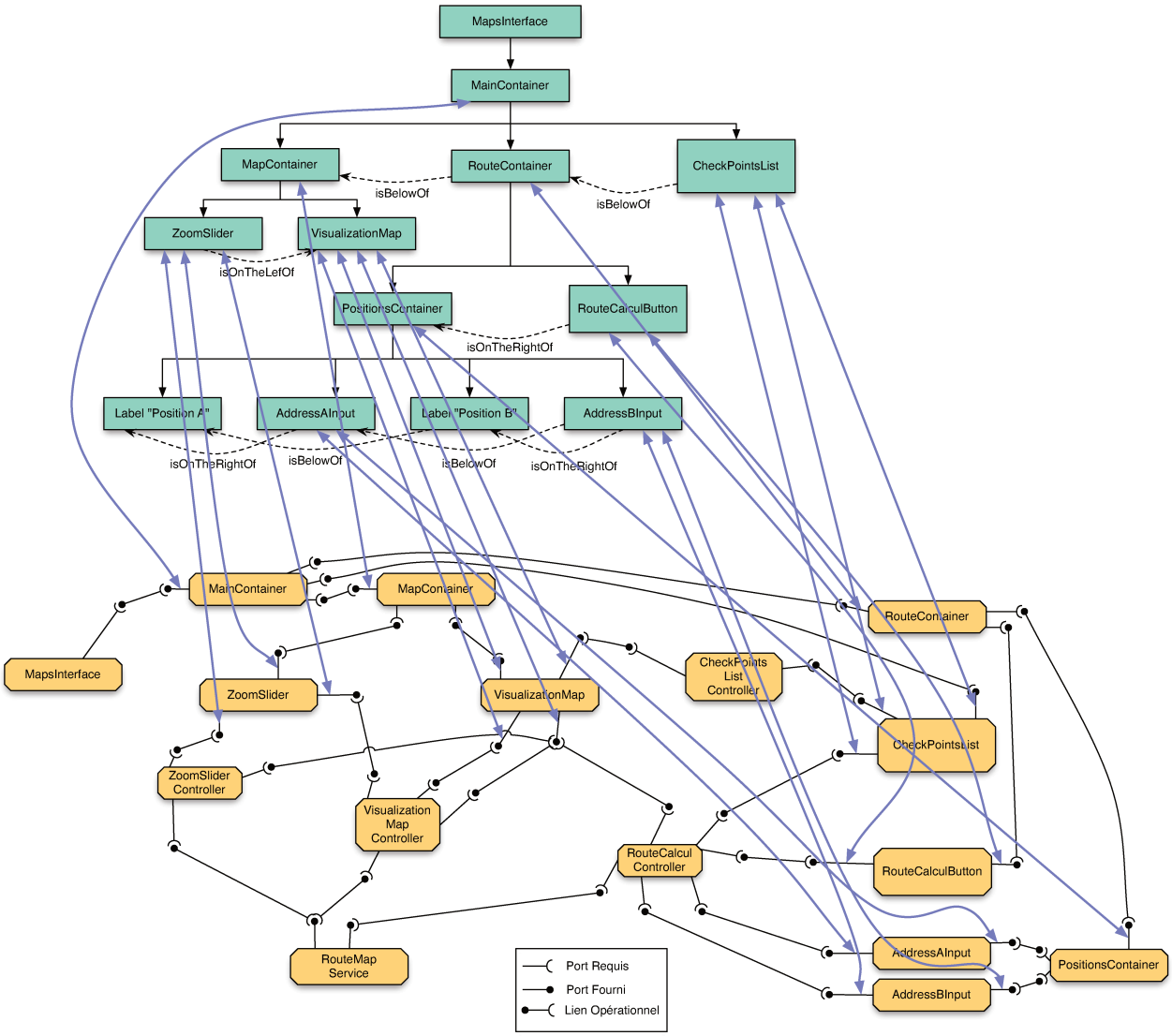



FIGURE A.14 – Liens entre modèle opérationnel et modèle de l'interface graphique de l'application "Maps".

ANNEXE B

# Ontologies définies pour les modèles de description d'une application

CETTE annexe expose les différentes ontologies que nous avons défini pour mettre en œuvre les modèles représentants les différents points de vues d'une application.

## B.1 UIOnto

*UIOnto* est l'ontologie pour décrire la partie graphique d'une application. Cette ontologie est constituée de 2 classes et 11 propriétés. Les 2 classes permettent de représenter la racine de l'interface graphique (*UIRoot*) sous classe de *UIElem* représentant les éléments graphiques. Une propriété permet de construire l'arbre de l'interface graphique en précisant les éléments graphiques contenus dans un autre (*containsUIElement*). Enfin, les propriétés restantes permettent de décrire le positionnement des éléments dans l'interface (*hasPosition, hasRelativeGridPositionTo, isAboveOf, isOnTheLeftOf, isOnTheRightOf, isBelowOf, isAboveLeftOf, isAboveRightOf, isBelowLeftOf, isBelowRightOf*).

Listing B.1 – **Ontologie au format OWL : fichier UIOnto.owl**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE rdf [
    <!ENTITY rdfs      "http://www.w3.org/2000/01/rdf-schema">
    <!ENTITY rdf       "http://www.w3.org/1999/02/22-rdf-syntax-ns">
    <!ENTITY owl       "http://www.w3.org/2002/07/owl">
    <!ENTITY taskonto      "http://rainbow.i3s.unice.fr/TaskOnto.owl">
    <!ENTITY ontocompo      "http://rainbow.i3s.unice.fr/OntoCompo.owl">
    <!ENTITY uionto      "http://rainbow.i3s.unice.fr/UIOnto.owl">
     ]>

<rdf:RDF
    xmlns:rdf ="&rdf;#"
    xmlns:rdfs="&rdfs;#"
    xmlns:owl ="&owl;#"
    xmlns:taskonto="&taskonto;#"
    xmlns:ontocompo="&ontocompo;#"
    xmlns:uionto="&uionto;#"
    xmlns      ="&rdfs;#"
    xml:base   ="&uionto;" >

    <!-- Définition des classes -->

    <Class rdf:ID="UIElem">
         <label xml:lang="en">UIElem</label>
         <comment xml:lang="en">null.</comment>
         <label xml:lang="fr">Elément d interface graphique</label>
         <comment xml:lang="fr">Définit un élément d interface graphique.</comment>
     </Class>

```

```
30      <Class rdf:ID="UIRoot">
31          <rdfs:subClassOf rdf:resource="#UIElem" />
32          <label xml:lang="en">UIRoot</label>
33          <comment xml:lang="en">null.</comment>
34          <label xml:lang="fr">Racine de l interface graphique.</label>
35          <comment xml:lang="fr">Définit la racine de l interface graphique.</comment>
36      </Class>
37
38      <!-- Définition des propriétés -->
39
40      <owl:TransitiveProperty rdf:ID="containsUIElement">
41          <domain rdf:resource="#UIElem"/>
42          <range rdf:resource="#UIElem"/>
43          <label xml:lang="en">contains UIElement</label>
44          <comment xml:lang="en">null.</comment>
45          <label xml:lang="fr">contient un élément de l interface graphique</label>
46          <comment xml:lang="fr">Permet de construire l arbre UI en définissant les éléments↩
                contenus dans un autre.</comment>
47      </owl:TransitiveProperty>
48
49      <!-- Définition des positions -->
50
51      <rdf:Property rdf:ID="hasPosition">
52          <domain rdf:resource="#UIElem"/>
53          <range rdf:resource="#UIElem"/>
54          <label xml:lang="en">has a position with</label>
55          <comment xml:lang="en">null.</comment>
56          <label xml:lang="fr">a un positionnement par rapport à</label>
57          <comment xml:lang="fr">Permet de définir la position d un intéracteur par rapport ↩
                à un autre.</comment>
58      </rdf:Property>
59
60      <!-- Positions relatives entre éléments graphiques -->
61
62      <rdf:Property rdf:ID="hasRelativeGridPositionTo">
63          <rdfs:subPropertyOf rdf:resource="#hasPosition" />
64          <label xml:lang="en">has a grid position compared with</label>
65          <comment xml:lang="en">null.</comment>
66          <label xml:lang="fr">grille de positionnement par rapport à</label>
67          <comment xml:lang="fr">Permet de définir la position d un intéracteur par rapport ↩
                à un autre lorsque celui ci est situé au centre d une grille.</comment>
68      </rdf:Property>
69
70      <owl:TransitiveProperty rdf:ID="isAboveOf">
71          <rdfs:subPropertyOf rdf:resource="#hasRelativeGridPositionTo" />
72          <label xml:lang="en">is on above position</label>
73          <comment xml:lang="en">null.</comment>
74          <label xml:lang="fr">est dans une position au dessus</label>
75          <comment xml:lang="fr">Permet de définir les positions d un intéracteur au dessus ↩
                d un autre intéracteur.</comment>
76      </owl:TransitiveProperty>
77
78      <owl:TransitiveProperty rdf:ID="isOnTheLeftOf">
79          <rdfs:subPropertyOf rdf:resource="#hasRelativeGridPositionTo" />
80          <label xml:lang="en">is on left position</label>
81          <comment xml:lang="en">null.</comment>
82          <label xml:lang="fr">est dans une position à gauche</label>
83          <comment xml:lang="fr">Permet de définir les positions d un intéracteur à gauche d↩
                un autre intéracteur.</comment>
84      </owl:TransitiveProperty>
85
86      <owl:TransitiveProperty rdf:ID="isOnTheRightOf">
87          <rdfs:subPropertyOf rdf:resource="#hasRelativeGridPositionTo" />
88          <label xml:lang="en">is on right position</label>
89          <comment xml:lang="en">null.</comment>
90          <label xml:lang="fr">est dans une position à droite</label>
```

```
91          <comment xml:lang="fr">Permet de définir les positions d un intéracteur à droite d↩
                un autre intéracteur.</comment>
92      </owl:TransitiveProperty>
93
94      <owl:TransitiveProperty rdf:ID="isBelowOf">
95          <rdfs:subPropertyOf rdf:resource="#hasRelativeGridPositionTo" />
96          <label xml:lang="en">is on below position</label>
97          <comment xml:lang="en">null.</comment>
98          <label xml:lang="fr">est dans une position en dessous</label>
99          <comment xml:lang="fr">Permet de définir les positions d un intéracteur en dessous↩
                d un autre intéracteur.</comment>
100     </owl:TransitiveProperty>
101
102     <owl:TransitiveProperty rdf:ID="isAboveLeftOf">
103         <rdfs:subPropertyOf rdf:resource="#hasRelativeGridPositionTo" />
104         <rdfs:subPropertyOf rdf:resource="#isAboveOf" />
105         <rdfs:subPropertyOf rdf:resource="#isOnTheLeftOf" />
106         <label xml:lang="en">above left</label>
107         <comment xml:lang="en">null.</comment>
108         <label xml:lang="fr">au-dessus à gauche</label>
109         <comment xml:lang="fr">au-dessus à gauche</comment>
110     </owl:TransitiveProperty>
111
112    <owl:TransitiveProperty rdf:ID="isAboveRightOf">
113         <rdfs:subPropertyOf rdf:resource="#hasRelativeGridPositionTo" />
114         <rdfs:subPropertyOf rdf:resource="#isAboveOf" />
115         <rdfs:subPropertyOf rdf:resource="#isOnTheRightOf" />
116         <label xml:lang="en">above right</label>
117         <comment xml:lang="en">null.</comment>
118         <label xml:lang="fr">au-dessus à droite</label>
119         <comment xml:lang="fr">au-dessus à droite.</comment>
120     </owl:TransitiveProperty>
121
122     <owl:TransitiveProperty rdf:ID="isBelowLeftOf">
123         <rdfs:subPropertyOf rdf:resource="#hasRelativeGridPositionTo" />
124         <rdfs:subPropertyOf rdf:resource="#isBelowOf" />
125         <rdfs:subPropertyOf rdf:resource="#isOnTheLeftOf" />
126         <label xml:lang="en">below left</label>
127         <comment xml:lang="en">null.</comment>
128         <label xml:lang="fr">en-dessous à gauche</label>
129         <comment xml:lang="fr">en-dessous à gauche</comment>
130     </owl:TransitiveProperty>
131
132     <owl:TransitiveProperty rdf:ID="isBelowRightOf">
133         <rdfs:subPropertyOf rdf:resource="#hasRelativeGridPositionTo" />
134         <rdfs:subPropertyOf rdf:resource="#isBelowOf" />
135         <rdfs:subPropertyOf rdf:resource="#isOnTheRightOf" />
136         <label xml:lang="en">below right</label>
137         <comment xml:lang="en">null.</comment>
138         <label xml:lang="fr">en-dessous à droite</label>
139         <comment xml:lang="fr">en-dessous à droite.</comment>
140     </owl:TransitiveProperty>
141
142 </rdf:RDF>
```

## B.2 TaskOnto

*TaskOnto* est l'ontologie pour décrire l'arbre de tâches d'une application. Elle est constituée de 6 classes dont la principale permet de décrire une Tâche de l'arbre (*Task*). Les 5 autres classes sont des sous-classes de celle-ci. Une classe permet de déterminer la Tâche racine de l'arbre (*RootTask*). Puis les 4 classes suivantes *AbstractTask, UserTask, SystemTask, InteractionTask* permettent de décrire respectivement les Tâches Abstraites, Utilisateurs, Systèmes et d'Intéractions. L'ontologie est

constitué également de 7 propriétés. *hasSubtask* et *hasParentTask* permet de décrire la hiérarchie de l'arbre de tâches. Les 5 autres propriétés permettent décrire les relations temporelles entre tâches avec *hasTemporalOperator* pour décrire un lien temporel entre deux tâches et 4 propriétés, sous-propriétés de celle-ci qui sont *isInParrallelWith, isInSequenceWith, enablingNext, isInAChoiceWith* décrivant respectivement le parallélisme entre deux tâches, la séquentialité avec un transfert d'information, la séquentialité sans transfert d'information et un choix entre deux tâches.

Listing B.2 – **Ontologie au format OWL : fichier TaskOnto.owl**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE rdf [
   <!ENTITY rdfs      "http://www.w3.org/2000/01/rdf-schema">
   <!ENTITY rdf       "http://www.w3.org/1999/02/22-rdf-syntax-ns">
   <!ENTITY owl       "http://www.w3.org/2002/07/owl">
   <!ENTITY taskonto      "http://rainbow.i3s.unice.fr/TaskOnto.owl">
   <!ENTITY ontocompo      "http://rainbow.i3s.unice.fr/OntoCompo.owl">
    ]>

<rdf:RDF
   xmlns:rdf ="&rdf;#"
   xmlns:rdfs="&rdfs;#"
   xmlns:owl ="&owl;#"
   xmlns:taskonto="&taskonto;#"
   xmlns:ontocompo="&ontocompo;#"
   xmlns     ="&rdfs;#"
   xml:base  ="&taskonto;" >

   <!-- Définition des classes -->

   <Class rdf:ID="Task">
        <label xml:lang="en">Task</label>
        <comment xml:lang="en">null.</comment>
        <label xml:lang="fr">Tâche</label>
        <comment xml:lang="fr">Définit une tâche c est-à-dire une action possible au ↩
            niveau de l application.</comment>
    </Class>

    <Class rdf:ID="AbstractTask">
        <rdfs:subClassOf rdf:resource="#Task" />
        <label xml:lang="en">AbstractTask</label>
        <comment xml:lang="en">null.</comment>
        <label xml:lang="fr">Tâche abstraite</label>
        <comment xml:lang="fr">Définit une tâche abstraite de l arbre de tâche.</comment>
    </Class>

    <Class rdf:ID="UserTask">
        <rdfs:subClassOf rdf:resource="#Task" />
        <label xml:lang="en">UserTask</label>
        <comment xml:lang="en">null.</comment>
        <label xml:lang="fr">Tâche utilisateur</label>
        <comment xml:lang="fr">Définit une tâche utilisateur de l arbre de tâche.</comment↩
            >
    </Class>

    <Class rdf:ID="SystemTask">
        <rdfs:subClassOf rdf:resource="#Task" />
        <label xml:lang="en">SystemTask</label>
        <comment xml:lang="en">null.</comment>
        <label xml:lang="fr">Tâche système</label>
        <comment xml:lang="fr">Définit une tâche système de l arbre de tâche.</comment>
    </Class>
```

```
51
52      <Class rdf:ID="InteractionTask">
53          <rdfs:subClassOf rdf:resource="#Task" />
54          <label xml:lang="en">InteractionTask</label>
55          <comment xml:lang="en">null.</comment>
56          <label xml:lang="fr">Tâche d intéraction</label>
57          <comment xml:lang="fr">Définit une tâche d intéraction de l arbre de tâche.</↩
                comment>
58      </Class>
59
60      <Class rdf:ID="RootTask">
61          <rdfs:subClassOf rdf:resource="#AbstractTask" />
62          <label xml:lang="en">RootTask</label>
63          <comment xml:lang="en">null.</comment>
64          <label xml:lang="fr">Tâche racine</label>
65          <comment xml:lang="fr">Définit la tâche racine de l arbre de tâche.</comment>
66      </Class>
67
68      <!-- Définition des propriétés -->
69
70      <owl:TransitiveProperty rdf:ID="hasSubtask">
71          <domain rdf:resource="#Task"/>
72          <range rdf:resource="#Task"/>
73          <label xml:lang="en">has subtask</label>
74          <comment xml:lang="en">null.</comment>
75          <label xml:lang="fr">a pour sous tâche</label>
76          <comment xml:lang="fr">Permet de définir les sous tâches d une tâche.</comment>
77
78          <owl:inverseOf rdf:resource="#hasParentTask"/>
79      </owl:TransitiveProperty>
80
81      <owl:TransitiveProperty rdf:ID="hasParentTask">
82          <domain rdf:resource="#Task"/>
83          <range rdf:resource="#Task"/>
84          <label xml:lang="en">has parent task</label>
85          <comment xml:lang="en">null.</comment>
86          <label xml:lang="fr">a pour tâche parente</label>
87          <comment xml:lang="fr">Permet de définir la tâche parente à la tâche courante.</↩
                comment>
88
89          <owl:inverseOf rdf:resource="#hasSubtask"/>
90      </owl:TransitiveProperty>
91
92      <rdf:Property rdf:ID="hasTemporalOperator">
93          <domain rdf:resource="#Task"/>
94          <range rdf:resource="#Task"/>
95          <label xml:lang="en">has temporal operator</label>
96          <comment xml:lang="en">null.</comment>
97          <label xml:lang="fr">associe l opérateur temporel</label>
98          <comment xml:lang="fr">Permet d associer un opérateur temporel à une tâche ce qui ↩
                définit la liaison entre les deux tâches liées.</comment>
99      </rdf:Property>
100
101     <owl:SymmetricProperty rdf:ID="isInParrallelWith">
102         <rdfs:subPropertyOf rdf:resource="#hasTemporalOperator" />
103         <label xml:lang="en">Parrallel Operator</label>
104         <comment xml:lang="en">null.</comment>
105         <label xml:lang="fr">Opérateur de parallèlisme</label>
106         <comment xml:lang="fr">Définit le parallélisme entre les tâches.</comment>
107     </owl:SymmetricProperty>
108
109     <owl:TransitiveProperty rdf:ID="isInSequenceWith">
110         <rdfs:subPropertyOf rdf:resource="#hasTemporalOperator" />
111         <label xml:lang="en">Sequential Operator</label>
112         <comment xml:lang="en">null.</comment>
113         <label xml:lang="fr">Opérateur de séquentialité</label>
```

```
114         <comment xml:lang="fr">Définit la séquentialité entre les tâches.</comment>
115     </owl:TransitiveProperty>
116
117     <owl:SymmetricProperty rdf:ID="isInAChoiceWith">
118         <rdfs:subPropertyOf rdf:resource="#hasTemporalOperator" />
119         <label xml:lang="en">Choice Operator</label>
120         <comment xml:lang="en">null.</comment>
121         <label xml:lang="fr">Opérateur de choix</label>
122         <comment xml:lang="fr">Définit le choix entre les tâches.</comment>
123     </owl:SymmetricProperty>
124
125     <owl:TransitiveProperty rdf:ID="enablingNext">
126         <rdfs:subPropertyOf rdf:resource="#hasTemporalOperator" />
127         <label xml:lang="en">Enabling Operator</label>
128         <comment xml:lang="en">null.</comment>
129         <label xml:lang="fr">Opérateur d actionnement</label>
130         <comment xml:lang="fr">Permet l actionnement de la tâche suivante.</comment>
131     </owl:TransitiveProperty>
132
133 </rdf:RDF>
```

## B.3 OntoCompo

*OntoCompo* est l'ontologie pour décrire le modèle opérationnel d'une application. Cette ontologie est constituée de 7 classes et 10 propriétés. *Application* et *AppElement* sont les deux classes permettant de décrire la constitution d'une application en terme d'éléments logiciels. Ainsi l'application va être "construite avec" ou constituée de plusieurs éléments logiciel, ce qui constitue une première propriété *isBuiltWith*. Comme décrit dans le chapitre présentant nos modèles, nos éléments d'applications vont être constitués (propriété *hasPort*) de ports (*Port*) qui peuvent être soit fournis, soit requis (classes *RequiredPort* et *ProvidedPort* sous classes de *Port*). La description de la connexion entre les ports des différents éléments logiciels de l'application est assurée par la propriété *connectedTo*. Chaque port va être associé (propriété *hasType*) à un Type (classe *Type*) dont 3 instances sont définies (*Trigger, Input, Output*). Chaque port joue un rôle (classe *Role*) particulier dans l'application et comme nous nous sommes concentrés sur les interactions graphiques, nous avons défini 2 instances qui sont *UIRole* et *UIComponentRole*. L'association entre un port et un rôle est effectuée grâce à la propriété *hasRole*.
Nous avons défini 2 propriétés : *hasID* qui permet d'associer un identifiant à chacun des éléments logiciels de l'application et *hasName* qui permet d'associer un nom à chaque port de chacun des éléments logiciels de l'application. Ces identifiants et noms sont définis par rapport aux composants fractals que nous avons décidé d'utiliser comme composants logiciels pour développer les applications à composer de notre prototype.
Enfin, nous avons définis 3 propriétés afin de lier les modèles entre eux comme défini dans le chapitre 3. La première *taskLinkedWithPort* permet de lier une tâche défini dans TaskOnto à un port d'un élément logiciel. La seconde *uiLinkedWithPort* permet de lier un élément graphique défini dans UIOnto à un port d'un élément logiciel. Et enfin, la dernière *taskLinkedWithUIElem* permet de lier une tâche et un élément graphique. Ces propriétés permettent d'établir des ponts entre les 3 modèles et forment les bases aux raisonnements que nous avons sur les liaisons entre les différentes entités constituants les différents points de vue de notre application.

Listing B.3 – **Ontologie au format OWL : fichier OntoCompo.owl**

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE rdf [
```

```
3     <!ENTITY rdfs      "http://www.w3.org/2000/01/rdf-schema">
4     <!ENTITY rdf       "http://www.w3.org/1999/02/22-rdf-syntax-ns">
5     <!ENTITY owl       "http://www.w3.org/2002/07/owl">
6     <!ENTITY taskonto      "http://rainbow.i3s.unice.fr/TaskOnto.owl">
7     <!ENTITY ontocompo      "http://rainbow.i3s.unice.fr/OntoCompo.owl">
8     <!ENTITY uionto      "http://rainbow.i3s.unice.fr/UIOnto.owl">
9      ]>
10
11  <rdf:RDF
12     xmlns:rdf ="&rdf;#"
13     xmlns:rdfs="&rdfs;#"
14     xmlns:owl ="&owl;#"
15     xmlns:taskonto="&taskonto;#"
16     xmlns:uionto="&uionto;#"
17     xmlns:ontocompo="&ontocompo;#"
18     xmlns     ="&rdfs;#"
19     xml:base  ="&ontocompo;" >
20
21     <!-- Définition des classes -->
22
23     <Class rdf:ID="Application">
24          <label xml:lang="en">Application</label>
25          <comment xml:lang="en">null.</comment>
26          <label xml:lang="fr">Application</label>
27          <comment xml:lang="fr">Définit la représentation d une application.</comment>
28     </Class>
29
30     <Class rdf:ID="AppElement">
31          <label xml:lang="en">AppElement</label>
32          <comment xml:lang="en">null.</comment>
33          <label xml:lang="fr">AppElement</label>
34          <comment xml:lang="fr">Elément d une application.</comment>
35     </Class>
36
37     <Class rdf:ID="Port">
38          <label xml:lang="en">Port</label>
39          <comment xml:lang="en">null.</comment>
40          <label xml:lang="fr">Port</label>
41          <comment xml:lang="fr">Port d un élément d une application.</comment>
42     </Class>
43
44     <Class rdf:ID="RequiredPort">
45          <subClassOf rdf:resource="#Port"/>
46          <label xml:lang="en">RequiredPort</label>
47          <comment xml:lang="en">null.</comment>
48          <label xml:lang="fr">Port requis</label>
49          <comment xml:lang="fr">Port requis d un élément d une application.</comment>
50     </Class>
51
52     <Class rdf:ID="ProvidedPort">
53          <subClassOf rdf:resource="#Port"/>
54          <label xml:lang="en">ProvidedPort</label>
55          <comment xml:lang="en">null.</comment>
56          <label xml:lang="fr">Port fournis</label>
57          <comment xml:lang="fr">Port fournis d un élément d une application.</comment>
58     </Class>
59
60     <Class rdf:ID="Role">
61          <label xml:lang="en">Role</label>
62          <comment xml:lang="en">null.</comment>
63          <label xml:lang="fr">Rôle</label>
64          <comment xml:lang="fr">Rôle d un port d un élément d une application.</comment>
65     </Class>
66
67     <Class rdf:ID="Type">
68          <label xml:lang="en">Type</label>
```

```
69          <comment xml:lang="en">null.</comment>
70          <label xml:lang="fr">Type</label>
71          <comment xml:lang="fr">Type d un port d un élément d une application.</comment>
72      </Class>
73
74      <!-- Définition des propriétés -->
75
76      <rdf:Property rdf:ID="isBuiltWith">
77          <domain rdf:resource="#Application"/>
78          <range rdf:resource="#AppElement"/>
79          <label xml:lang="en">isBuiltWith</label>
80          <comment xml:lang="en">null.</comment>
81          <label xml:lang="fr">est construit avec</label>
82          <comment xml:lang="fr">Permet d attacher un élémént d application avec son ↩
                application.</comment>
83      </rdf:Property>
84
85      <rdf:Property rdf:ID="hasPort">
86          <domain rdf:resource="#AppElement"/>
87          <range rdf:resource="#Port"/>
88          <label xml:lang="en">hasPort</label>
89          <comment xml:lang="en">null.</comment>
90          <label xml:lang="fr">a un port</label>
91          <comment xml:lang="fr">Permet d attacher un port d application à un élément d ↩
                application.</comment>
92      </rdf:Property>
93
94      <owl:SymmetricProperty rdf:ID="connectedTo">
95          <domain rdf:resource="#Port"/>
96          <range rdf:resource="#Port"/>
97          <label xml:lang="en">connected to</label>
98          <comment xml:lang="en">null.</comment>
99          <label xml:lang="fr">est connecté à</label>
100         <comment xml:lang="fr">Permet de connecter deux ports entre eux.</comment>
101     </owl:SymmetricProperty>
102
103     <rdf:Property rdf:ID="hasType">
104         <domain rdf:resource="#Port"/>
105         <range rdf:resource="#Type"/>
106         <label xml:lang="en">has type</label>
107         <comment xml:lang="en">null.</comment>
108         <label xml:lang="fr">a pour type</label>
109         <comment xml:lang="fr">Permet d attacher un type à un port d un élément de l ↩
                application.</comment>
110     </rdf:Property>
111
112     <rdf:Property rdf:ID="hasRole">
113         <domain rdf:resource="#Port"/>
114         <range rdf:resource="#Role"/>
115         <label xml:lang="en">has role</label>
116         <comment xml:lang="en">null.</comment>
117         <label xml:lang="fr">a pour rôle</label>
118         <comment xml:lang="fr">Permet d attacher un rôle à un port d un élément de l ↩
                application.</comment>
119     </rdf:Property>
120
121     <owl:SymmetricProperty rdf:ID="taskLinkedWithPort">
122         <domain rdf:resource="&taskonto;#Task"/>
123         <range rdf:resource="#Port"/>
124         <label xml:lang="en">task linked with port</label>
125         <comment xml:lang="en">null.</comment>
126         <label xml:lang="fr">la tâche est liée au port</label>
127         <comment xml:lang="fr">Permet de lier une tâche à un port d un élément de l ↩
                application.</comment>
128     </owl:SymmetricProperty>
129
```

```
130     <owl:SymmetricProperty rdf:ID="uiLinkedWithPort">
131         <domain rdf:resource="&uionto;#UIElem"/>
132         <range rdf:resource="#Port"/>
133         <label xml:lang="en">ui element linked with port</label>
134         <comment xml:lang="en">null.</comment>
135         <label xml:lang="fr">l élément UI est lié au port</label>
136         <comment xml:lang="fr">Permet de lier un élément d interface graphique à un port d↩
                un élément de l application.</comment>
137     </owl:SymmetricProperty>
138 
139     <owl:SymmetricProperty rdf:ID="taskLinkedWithUIElem">
140         <domain rdf:resource="&taskonto;#Task"/>
141         <range rdf:resource="&uionto;#UIElem"/>
142         <label xml:lang="en">task linked with ui element</label>
143         <comment xml:lang="en">null.</comment>
144         <label xml:lang="fr">la tâche est liée à l élément UI</label>
145         <comment xml:lang="fr">Permet de lier une tâche à un élément UI de l interface ↩
              graphique de l application.</comment>
146     </owl:SymmetricProperty>
147 
148     <rdf:Property rdf:ID="hasID">
149         <domain rdf:resource="#AppElement"/>
150         <label xml:lang="en">has id</label>
151         <comment xml:lang="en">null.</comment>
152         <label xml:lang="fr">a pour identifiant</label>
153         <comment xml:lang="fr">Permet de définir l identifiant du composant fractal ↩
              représenté par l élément de l application.</comment>
154     </rdf:Property>
155 
156     <rdf:Property rdf:ID="hasName">
157         <domain rdf:resource="#Port"/>
158         <label xml:lang="en">has name</label>
159         <comment xml:lang="en">null.</comment>
160         <label xml:lang="fr">a pour nom</label>
161         <comment xml:lang="fr">Permet de définir le nom donné au port d un élément de l ↩
              application.</comment>
162     </rdf:Property>
163 
164 </rdf:RDF>
```

ANNEXE C

# Descriptions sémantiques de l'application "Maps"

CETTE annexe expose une partie des descriptions sémantiques utilisées dans le prototype pour l'application "Maps".

## C.1 MapsUI.rdf.part

L'extrait de la description sémantique présenté ci-dessous, permet de décrire le formulaire présent dans l'interface de l'application "Maps", c'est-à-dire les deux entrées textes pour pouvoir renseigner l'adresse de départ et l'adresse d'arrivée, ainsi que le bouton permettant de déclencher le calcul de l'itinéraire entre les deux adresses renseignées (cf. figure C.1). Les positions des éléments graphiques dans l'interface sont décrits et des liens avec le modèle opérationnel de l'application Maps sont effectués.

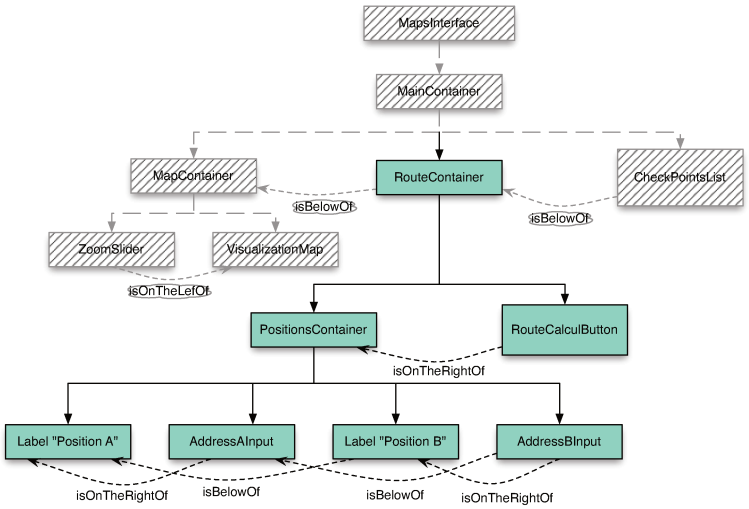



FIGURE C.1 – Sous-partie du modèle de l'interface graphique de l'application Maps.

Listing C.1 – **Extrait de la description sémantique au format RDF : fichier MapsUi.rdf.part**

```
1 <UIElem rdf:ID="RouteContainer">
2 <!-- Description de cet élément graphique qui contient 2 éléments: le "conteneur" avec les↩
      entrées textes et le bouton -->
3     <containsUIElement rdf:resource="#PositionsContainer" />
4     <containsUIElement rdf:resource="#RouteCalculButton" />
5
```

```
6  <!-- Ce "conteneur" est situé au dessus du "conteneur" de la carte -->
7      <isBelowOf rdf:resource="#MapContainer" />
8
9  <!-- Un lien est établi avec un port d un élément logiciel du modèle opérationnel: Ce port↩
       permet de récupérer l élément graphique, en l occurance dans ce prototype, il s agit ↩
       d un composant SWING -->
10         <ontocompo:uiLinkedWithPort rdf:resource="&ontocompo;-instances#GetRouteContainer"↩
               />
11 </UIElem>
12
13     <UIElem rdf:ID="PositionsContainer">
14 <!-- Description de cet élément graphique qui contient 4 éléments: l adresse de départ, l ↩
       adresse d arrivée et les labels associés -->
15         <containsUIElement rdf:resource="#LabelPositionA" />
16         <containsUIElement rdf:resource="#AddressAInput" />
17         <containsUIElement rdf:resource="#LabelPositionB" />
18         <containsUIElement rdf:resource="#AddressBInput" />
19
20 <!-- Un lien est établi avec un port d un élément logiciel du modèle opérationnel: Ce port↩
       permet de récupérer l élément graphique, en l occurance dans ce prototype, il s agit ↩
       d un composant SWING -->
21         <ontocompo:uiLinkedWithPort rdf:resource="&ontocompo;-instances#↩
               GetPositionsContainer" />
22     </UIElem>
23
24         <UIElem rdf:ID="LabelPositionA">
25 <!-- Décrit le label de l élément graphique permettant de renseigner l adresse de départ ↩
       -->
26         </UIElem>
27
28         <UIElem rdf:ID="AddressAInput">
29 <!-- Décrit l élément graphique permettant de renseigner l adresse de départ -->
30 <!-- Il est situé à droite de son label -->
31             <isOnTheRightOf rdf:resource="#LabelPositionA" />
32
33 <!-- Deux liens sont établis avec deux ports du même élément logiciel du modèle ↩
       opérationnel -->
34 <!-- Le premier lien est celui vers le port "UI" "INPUT" qui défini une intéraction avec l↩
       utilisateur -->
35             <ontocompo:uiLinkedWithPort rdf:resource="&ontocompo;-instances#GetAddressFrom↩
                   " />
36 <!-- Le second lien est celui vers le port "UIComponent" qui permet de récupérer l élément↩
       graphique associé (composant SWING) -->
37             <ontocompo:uiLinkedWithPort rdf:resource="&ontocompo;-instances#↩
                   GetAddressAInput" />
38         </UIElem>
39
40         <UIElem rdf:ID="LabelPositionB">
41 <!-- Décrit le label de l élément graphique permettant de renseigner l adresse d arrivée ↩
       -->
42 <!-- Il est situé au dessus du label de l adresse de départ -->
43             <isBelowOf rdf:resource="#LabelPositionA" />
44         </UIElem>
45
46         <UIElem rdf:ID="AddressBInput">
47 <!-- Décrit l élément graphique permettant de renseigner l adresse d arrivée -->
48 <!-- Il est situé à droite de son label et en dessous de l adresse de départ -->
49             <isBelowOf rdf:resource="#AddressAInput" />
50             <isOnTheRightOf rdf:resource="#LabelPositionB" />
51
52 <!-- Deux liens sont établis avec deux ports du même élément logiciel du modèle ↩
       opérationnel comme pour l adresse de départ -->
53             <ontocompo:uiLinkedWithPort rdf:resource="&ontocompo;-instances#GetAddressTo" ↩
                   />
54             <ontocompo:uiLinkedWithPort rdf:resource="&ontocompo;-instances#↩
                   GetAddressBInput" />
```

```
55          </UIElem>
56
57      <UIElem rdf:ID="RouteCalculButton">
58  <!-- Décrit l élément graphique permettant de déclencher le calcul de l itinéraire -->
59  <!-- Il est situé à droite du "conteneur" des entrées textes -->
60          <isOnTheRightOf rdf:resource="#PositionsContainer" />
61
62  <!-- Deux liens sont établis avec deux ports du même élément logiciel du modèle ↩
        opérationnel: toujours un lien vers un port permettant de récupérer l élément ↩
        graphique (composant SWING) et le premier lien est vers le port "TRIGGER" permettant ↩
        de déclencher le calcul de l itinéraire -->
63          <ontocompo:uiLinkedWithPort rdf:resource="&ontocompo;-instances#↩
                ButtonActionPerformed" />
64          <ontocompo:uiLinkedWithPort rdf:resource="&ontocompo;-instances#↩
                GetRouteCalculButton" />
65      </UIElem>
```

## C.2 MapsTasks.rdf.part

L'extrait de la description sémantique de l'arbre de tâches de l'application présenté ci-dessous, permet de décrire l'ensemble des tâches attachées à la sous-partie de l'application choisie (cf. figure C.2). La hiérarchie des tâches est décrite, ainsi que les opérateurs temporels entre tâches. Certaines tâches vont avoir deux types de liens inter-modèles, le premier est un lien avec le modèle opérationnel de l'application tandis que le second lien est avec le modèle de l'interface graphique de l'application.

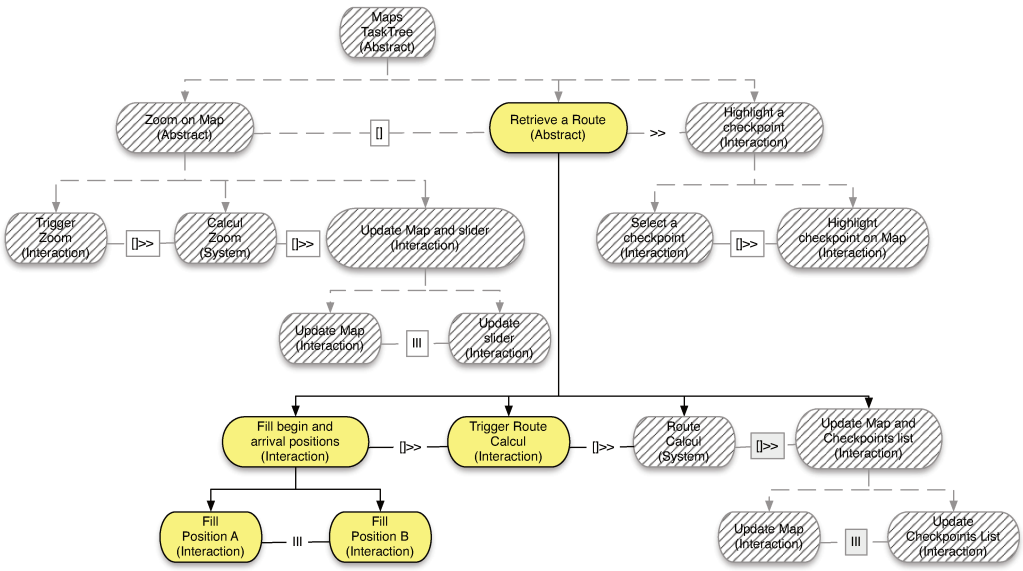



FIGURE C.2 – Sous-partie du modèle de tâches de l'application Maps.

Listing C.2 – **Extrait de la description sémantique au format RDF : fichier MapsTasks.rdf.part**

```
1  <AbstractTask rdf:ID="RetrieveARoute">
2      <hasParentTask rdf:resource="#MapsTaskTree" />
3
```

```
4  <!-- Déclaration des sous-tâches constituants la tâche permettant de remplir le formulaire↩
       et de lancer l exécution du calcul de l itinéraire. -->
5      <hasSubtask rdf:resource="#FillBeginAndArrivalPositions" />
6      <hasSubtask rdf:resource="#TriggerRouteCalcul" />
7      <hasSubtask rdf:resource="#RouteCalcul" />
8      <hasSubtask rdf:resource="#UpdateMapAndCheckpointsList" />
9
10     <enablingNext rdf:resource="#HighlightACheckpoint" />
11 </AbstractTask>
12
13     <InteractionTask rdf:ID="FillBeginAndArrivalPositions">
14         <hasParentTask rdf:resource="#RetrieveARoute" />
15
16 <!-- Déclaration des sous-tâches constituants la tâche permettant de remplir le formulaire↩
       . -->
17         <hasSubtask rdf:resource="#FillPositionA" />
18         <hasSubtask rdf:resource="#FillPositionB" />
19
20         <isInSequenceWith rdf:resource="#TriggerRouteCalcul" />
21     </InteractionTask>
22
23         <InteractionTask rdf:ID="FillPositionA">
24 <!-- Description de la tâche permettant de remplir l adresse de départ. -->
25             <hasParentTask rdf:resource="#FillBeginAndArrivalPositions" />
26
27 <!-- Description du parallélisme entre cette tâche et la tâche permettant de remplir l ↩
       adresse d arrivée. -->
28             <isInParrallelWith rdf:resource="#FillPositionB" />
29
30 <!-- Description des liens inter-modèles, d une part avec le modèle opérationnel et d ↩
       autre part avec le modèle de l interface graphique. -->
31             <ontocompo:taskLinkedWithPort rdf:resource="&ontocompo;-instances#↩
                   GetAddressFrom" />
32             <ontocompo:taskLinkedWithUIElem rdf:resource="&uionto;-instances#AddressAInput↩
                   " />
33         </InteractionTask>
34
35         <InteractionTask rdf:ID="FillPositionB">
36             <hasParentTask rdf:resource="#FillBeginAndArrivalPositions" />
37
38             <ontocompo:taskLinkedWithPort rdf:resource="&ontocompo;-instances#GetAddressTo↩
                   " />
39             <ontocompo:taskLinkedWithUIElem rdf:resource="&uionto;-instances#AddressBInput↩
                   " />
40         </InteractionTask>
41
42     <InteractionTask rdf:ID="TriggerRouteCalcul">
43 <!-- Description de la tâche permettant de déclencher le calcul de l itinéraire. -->
44         <hasParentTask rdf:resource="#RetrieveARoute" />
45
46         <isInSequenceWith rdf:resource="#RouteCalcul" />
47
48         <ontocompo:taskLinkedWithPort rdf:resource="&ontocompo;-instances#↩
               ButtonActionPerformed" />
49         <ontocompo:taskLinkedWithUIElem rdf:resource="&uionto;-instances#RouteCalculButton↩
               " />
50     </InteractionTask>
```

## C.3 MapsApp.rdf.part

Ci-dessous est présenté un extrait de la description sémantique du modèle opérationnel de l'application "Maps". Cet extrait est la sous partie des éléments opérationnels (cf. figure C.3) liés à la sous partie de l'interface graphique présentée ci-dessus. Effectivement, il représente les éléments logiciels

dont certains ports sont liés aux éléments graphiques. Chaque élément logiciel a un identifiant qui correspond dans le prototype à un composant fractal constitutif de l'application. De même, chaque port a un nom qui correspond au nom donné aux ports d'un composant fractal.

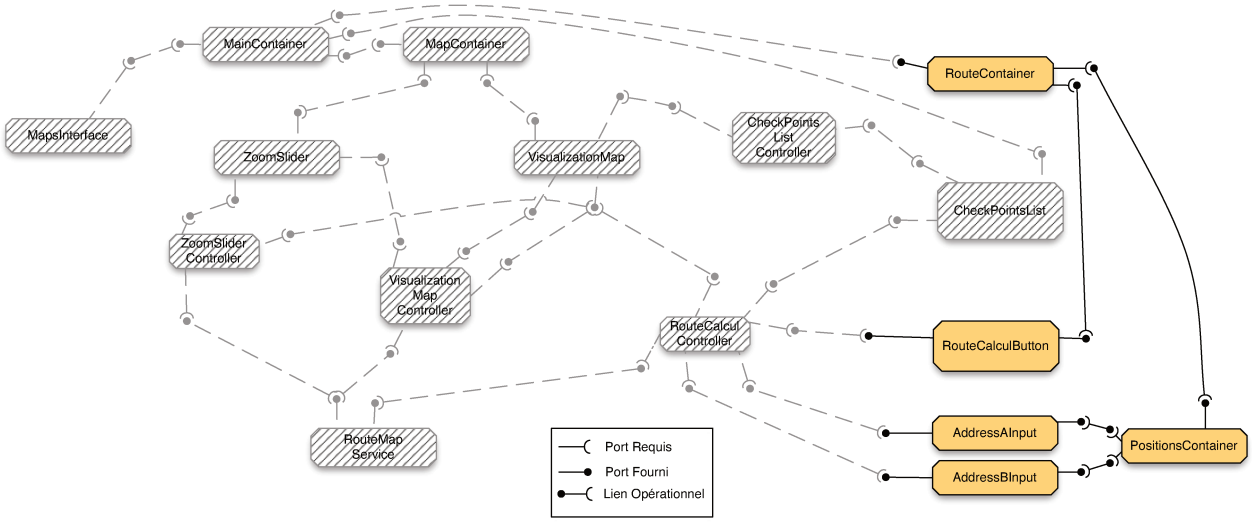



FIGURE C.3 – Sous-partie du modèle opérationnel de l'application Maps.

Listing C.3 – **Extrait de la description sémantique au format RDF : fichier MapsApp.rdf.part**

```
<Application rdf:ID="Maps">
   <rdfs:label xml:lang="en">Maps</rdfs:label>

<!-- Déclaration des éléments logiciels constituants l application -->
   <isBuiltWith ...

   <isBuiltWith rdf:resource="#PanelCentralComposant" />
   <isBuiltWith rdf:resource="#InputComposantFrom" />
   <isBuiltWith rdf:resource="#ButtonComposant" />
   <isBuiltWith rdf:resource="#ButtonComposant" />
   <isBuiltWith rdf:resource="#ButtonComposant" />

   <isBuiltWith ...
</Application>

<AppElement rdf:ID="RouteContainer">
<!-- Description des ports de l élément logiciel correspondant au "conteneur" de l adresse↩
     de départ, d arrivée et du bouton. -->
    <hasPort rdf:resource="#GetRouteContainer" />
    <hasPort rdf:resource="#RouteContainerGetPositionsContainer" />
    <hasPort rdf:resource="#RouteContainerGetRouteCalculButton" />

    <hasID>maps.graphics.RouteContainer</hasID>
</AppElement>

<!-- Port permettant de récupérer l élément graphique (composant SWING dans le cadre du ↩
    prototype) -->
    <ProvidedPort rdf:ID="GetRouteContainer">
        <hasType rdf:resource="&ontocompo;-instances#Input" />
        <hasRole rdf:resource="&ontocompo;-instances#UIComponentRole" />
        <connectedTo rdf:resource="#MainContainerGetRouteContainer" />
```

```
30
31          <hasName>pc</hasName>
32      </ProvidedPort>
33
34      <RequiredPort rdf:ID="RouteContainerGetPositionsContainer">
35          <hasType rdf:resource="&ontocompo;-instances#Input" />
36          <connectedTo rdf:resource="#GetPositionsContainer" />
37
38          <hasName>pc</hasName>
39      </RequiredPort>
40
41      <RequiredPort rdf:ID="RouteContainerGetRouteCalculButton">
42          <hasType rdf:resource="&ontocompo;-instances#Input" />
43          <connectedTo rdf:resource="#GetRouteCalculButton" />
44
45          <hasName>b</hasName>
46      </RequiredPort>
47
48  <AppElement rdf:ID="PositionsContainer">
49  <!-- Description des ports de l élément logiciel correspondant au "conteneur" de l adresse↩
        de départ, d arrivée. -->
50      <hasPort rdf:resource="#GetPositionsContainer" />
51      <hasPort rdf:resource="#PositionsContainerGetAddressAInput" />
52      <hasPort rdf:resource="#PositionsContainerGetAddressBInput" />
53
54      <hasID>maps.graphics.PositionsContainer</hasID>
55  </AppElement>
56
57  <!-- Port permettant de récupérer l élément graphique (composant SWING dans le cadre du ↩
       prototype) -->
58      <ProvidedPort rdf:ID="GetPositionsContainer">
59          <hasType rdf:resource="&ontocompo;-instances#Input" />
60          <hasRole rdf:resource="&ontocompo;-instances#UIComponentRole" />
61          <connectedTo rdf:resource="#RouteContainerGetPositionsContainer" />
62
63          <hasName>pc</hasName>
64      </ProvidedPort>
65
66      <RequiredPort rdf:ID="PositionsContainerGetAddressAInput">
67          <hasType rdf:resource="&ontocompo;-instances#Input" />
68          <connectedTo rdf:resource="#GetAddressAInput" />
69
70          <hasName>ifrom</hasName>
71      </RequiredPort>
72
73      <RequiredPort rdf:ID="PositionsContainerGetAddressBInput">
74          <hasType rdf:resource="&ontocompo;-instances#Input" />
75          <connectedTo rdf:resource="#GetAddressBInput" />
76
77          <hasName>ito</hasName>
78      </RequiredPort>
79
80  <AppElement rdf:ID="AddressAInput">
81  <!-- Description des ports de l élément logiciel correspondant à l adresse de départ -->
82      <hasPort rdf:resource="#GetAddressAInput" />
83      <hasPort rdf:resource="#GetAddressFrom" />
84
85      <hasID>maps.graphics.AddressAInput</hasID>
86  </AppElement>
87
88      <ProvidedPort rdf:ID="GetAddressAInput">
89          <hasType rdf:resource="&ontocompo;-instances#Input" />
90          <hasRole rdf:resource="&ontocompo;-instances#UIComponentRole" />
91          <connectedTo rdf:resource="#PositionsContainerGetAddressAInput" />
92
93          <hasName>ifrom</hasName>
```

```
94      </ProvidedPort>
95
96
97      <ProvidedPort rdf:ID="GetAddressFrom">
98          <hasType rdf:resource="&ontocompo;-instances#Input" />
99          <hasRole rdf:resource="&ontocompo;-instances#UIRole" />
100         <connectedTo rdf:resource="#RouteCalculControllerGetAddressFrom" />
101
102         <hasName>getFrom</hasName>
103     </ProvidedPort>
104
105 <AppElement rdf:ID="AddressBInput">
106 <!-- Description des ports de l élément logiciel correspondant à l adresse d arrivée -->
107     <hasPort rdf:resource="#GetAddressBInput" />
108     <hasPort rdf:resource="#GetAddressTo" />
109
110     <hasID>maps.graphics.AddressBInput</hasID>
111 </AppElement>
112
113 <!-- . . . Description des ports de AddressBInput . . . -->
114
115 <AppElement rdf:ID="RouteCalculButton">
116 <!-- Description des ports de l élément logiciel correspondant au bouton -->
117     <hasPort rdf:resource="#GetRouteCalculButton" />
118     <hasPort rdf:resource="#ButtonActionPerformed" />
119
120     <hasID>maps.graphics.ButtonComposant</hasID>
121 </AppElement>
122
123     <ProvidedPort rdf:ID="GetRouteCalculButton">
124         <hasType rdf:resource="&ontocompo;-instances#Input" />
125         <hasRole rdf:resource="&ontocompo;-instances#UIComponentRole" />
126         <connectedTo rdf:resource="#RouteContainerGetRouteCalculButton" />
127
128         <hasName>b</hasName>
129     </ProvidedPort>
130
131     <ProvidedPort rdf:ID="ButtonActionPerformed">
132         <hasType rdf:resource="&ontocompo;-instances#Trigger" />
133         <hasRole rdf:resource="&ontocompo;-instances#UIRole" />
134         <connectedTo rdf:resource="#RouteCalculControllerSearchRoute" />
135
136         <hasName>trigger</hasName>
137     </ProvidedPort>
```

ANNEXE D

# Réalisation du scénario de composition entre "Cinema" et "Maps"

CETTE annexe présente les différentes manipulations nécessaires pour réaliser la composition entre "Cinema" et "Maps".

## Première étape : sélection des éléments nécessaires

Les éléments graphiques qui doivent être sélectionnés dans les deux applications sont ceux présentés dans la figure D.1.
Pour l'application "Cinema", tous les éléments graphiques doivent être sélectionnés à l'exception de la liste décrivant les séances prévues pour la salle de cinéma sélectionnée.
Pour l'application "Maps", tous les éléments du formulaire, et la carte doivent être sélectionnés.

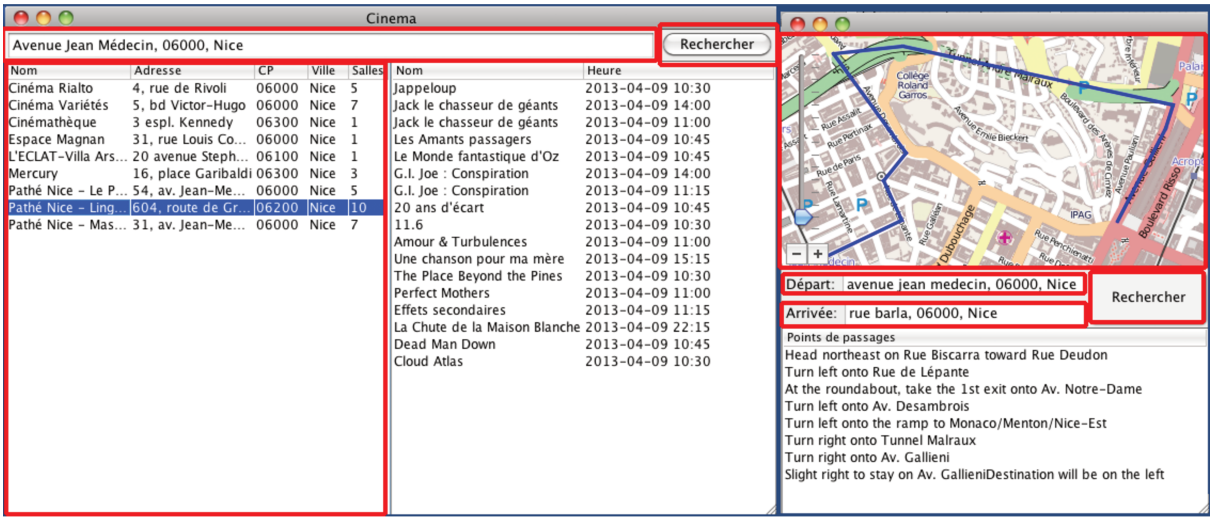



FIGURE D.1 – Illustration des éléments graphiques à sélectionner dans les deux applications "Cinema" et "Maps".

## Deuxième étape : substitutions successives

Différentes substitutions doivent être effectuées pour obtenir la composition désirée.
La substitution de l'entrée texte permettant de renseigner l'adresse de départ dans l'application "Maps" par l'entrée texte permettant de renseigner son adresse dans l'application "Cinema" doit être effectuée. Cette substitution permet de prendre pour adresse de départ de l'itinéraire, l'adresse renseignée par l'utilisateur.

La substitution de l'entrée texte permettant de renseigner l'adresse d'arrivée dans l'application "Maps" par la liste des salles de cinémas de l'application "Cinema" doit être effectuée. Cette substitution permettra de prendre pour adresse d'arrivée de l'itinéraire, l'adresse de la salle de cinéma sélectionnée dans la liste.
Enfin, la substitution du bouton du calcul de l'itinéraire dans l'application "Maps" par la liste des salles de cinémas de l'application "Cinema" doit être effectuée. Cette substitution permet de déclencher le calcul de l'itinéraire automatiquement lors de la sélection d'une salle de cinéma dans la liste.

## Troisième étape : placement

Le placement des éléments graphiques restants doit être effectué afin d'être le plus proche du schéma fourni lors des tests utilisateurs. Un exemple de ce placement est illustré sur la figure D.2.

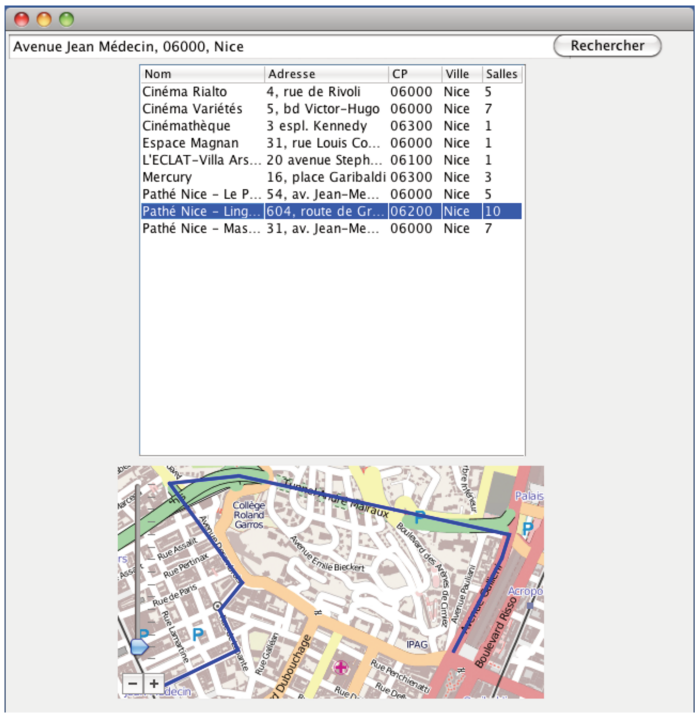



FIGURE D.2 – Illustration du placement des éléments graphiques de l'application résultante de la composition entre les deux applications "Cinema" et "Maps".

ANNEXE E

# Adapteur généré sous la forme d'un composant lors d'une substitution

CETTE annexe présente l'adapteur généré lors de la substitution illustrée sur la figure 5.6 page 102.

Listing E.1 – **Code Source Java : fichier AdapterCinemaGetAddressMapsGetAddressFrom.java**

```
1  package ontocompo.compo.cinemamaps;
2
3  import ontocompo.api.IGraphicElement;
4
5  import org.objectweb.fractal.fraclet.annotations.Component;
6  import org.objectweb.fractal.fraclet.annotations.Interface;
7  import org.objectweb.fractal.fraclet.annotations.Requires;
8
9  @Component(provides={@Interface(name="itext",signature=cinema.services.IInputGet.class),
       @Interface(name="getFrom",signature=maps.services.IInputGet.class)}) //Interface
       fournie par Cinema: port GetInput, Interface fournie par Maps: GetInputFrom
10 public class AdapterCinemaGetAddressMapsGetAddressFrom extends ontocompo.adapter.lib.
       Adapter implements cinema.services.IInputGet, maps.services.IInputGet, ontocompo.api.
       InitComponentItf {
11
12     @Requires(name="itext")
13     IInputGet input; // Le meme requis que le port connecte au port Cinema Get Input
14
15
16     public AdapterCinemaGetAddressMapsGetAddressFrom() {
17         // TODO Auto-generated method stub
18     }
19
20     @Override
21     public String getInput() {
22         // TODO Auto-generated method stub
23     }
24
25     @Override
26     public String getInput() {
27         // TODO Auto-generated method stub
28     }
29
30     public void init() {
31         // Nothing TODO !
32     }
33
34     public String getUid() {
35         return "ontocompo.compo.cinemamaps.AdapterCinemaGetAddressMapsGetAddressFrom";
36     }
37
38 }
```

ANNEXE F

# Déroulement du test utilisateur

CETTE annexe présente les étapes constituants le test utilisateur. L'entretien dure environ une heure et se déroule comme suit :

1. L'utilisateur est invité à décrire son profil de développeur afin que nous puissions confirmer sa catégorie d'appartenance. Des questions plus précises sur l'utilisation d'outils pour la composition d'applications (composition d'interfaces graphiques, compositions de services etc...) sont demandées ainsi que sur la connaissance et l'utilisation d'outils comme Yahoo Pipes ou IFTTT.com (cf. chapitre 2).
2. Le testeur explique à l'utilisateur le fonctionnement de l'outil et lui précise que durant le test, il lui sera demandé de réaliser plusieurs fois la même tâche ou des tâches similaires mais avec des connaissances supplémentaires.
3. Le scénario de composition d'applications est présenté à l'utilisateur. Une démonstration du fonctionnement des applications est effectuée.
   La maquette présentée sur la figure F.1 est présentée à l'utilisateur comme le résultat attendu de la composition.
4. Le testeur demande ensuite à l'utilisateur d'effectuer la sélection des éléments des deux interfaces nécessaire à la composition demandée. L'objectif est de faire manipuler le prototype par l'utilisateur.
5. Une fois la sélection effectuée, le testeur montre l'étape de "substitutions" en expliquant à l'utilisateur que la sélection effectuée jusqu'à présent est une sélection qui ne permet pas de lier les applications entre elles. Le testeur explique aussi que la substitution permet d'effectivement lier les applications entre elles, par substitutions d'éléments graphiques. Un élément doit être sélectionné et déclaré comme "gardé" puis un second élément doit être sélectionné et déclaré comme "supprimé", puis la substitution peut ensuite être déclenchée. Cette manipulation est alors montrée à l'utilisateur.
6. Si nécessaire, le testeur revient alors à l'étape de sélection. Il demande alors à l'utilisateur de refaire la sélection des éléments graphiques qui lui semblent nécessaires pour réaliser la composition demandée en réfléchissement à l'étape de substitutions (suite de la sélection).
7. Ensuite, le testeur invite l'utilisateur à effectuer les substitutions, celles-ci s'effectuant par paire d'éléments graphiques.
8. Le testeur demande ensuite à l'utilisateur de manipuler les éléments graphiques restants afin de les placer.
9. Cette étape complétée, le testeur explique qu'à la fin de cette étape de placement, l'application résultat peut être générée. Certains fichiers générés doivent être complétés afin de configurer l'application pour qu'elle ait le comportement voulu.
10. Mais avant de considérer cette génération, le testeur indique à l'utilisateur qu'il s'agit maintenant de revenir sur la sélection. Le testeur relance l'outils et replace l'utilisateur au niveau de

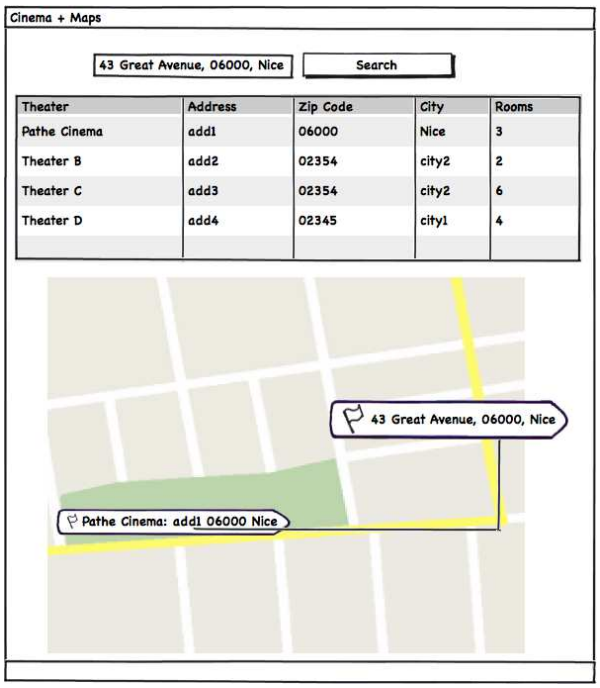



FIGURE F.1 – Maquette du résultat attendu pour la composition entre les applications "Cinema" et "Maps".

l'étape de Sélection. Le testeur dévoile alors les outils d'aide à la sélection (les extensions de sélection). Le testeur présente les trois types d'extensions, le premier s'appuyant sur la mise en page (le layout) des éléments graphiques des interfaces des applications, le second s'appuyant sur les tâches (actions possibles) associées à l'application et le troisième s'appuyant sur les éléments logiciels (sur l'assemblage de composants de l'application). Le premier type d'extension est découpé en deux parties, la partie qui s'appuie sur la structure hiérarchique de l'interface graphique de l'application et la seconde partie s'appuyant sur le placement des éléments graphiques les uns par rapport aux autres.

11. Le testeur propose à l'utilisateur d'effectuer une étape de familiarisation avec les extensions c'est-à-dire de manipuler les différentes extensions.
12. Puis, il demande à l'utilisateur de refaire la sélection en utilisant une ou plusieurs des extensions disponibles, voire aucune.
13. Une fois cette nouvelle sélection effectuée, le testeur demande à l'utilisateur si le fait d'avoir manipuler les interfaces des applications à travers les extensions change sa compréhension de l'étape de substitutions (question ouverte).
14. Le testeur revient de nouveau sur les extensions, et pour chaque type d'extension propose à l'utilisateur différents schémas représentants les différentes "vues" disponibles pour une application : un schéma représentant le modèle hiérarchique de l'interface de l'application ainsi que

le positionnement des éléments graphiques les uns par rapport aux autres (cf. figure A.2 et figure A.9 de l'annexe A), un schéma représentant l'arbre de tâches associé à une application (cf. figure A.4 et figure A.11 de l'annexe A), enfin un schéma représentant l'assemblage de composants de l'application (cf. figure A.3 et figure A.10 de l'annexe A).

Pour chaque schéma, c'est-à-dire pour chaque information supplémentaire que l'outil de composition pourrait fournir à l'utilisateur, le testeur demande à l'utilisateur s'il comprend le schéma, si le schéma lui serait utile et comment il utiliserait ces nouvelles informations.

15. Après cette phase de discussion autour des extensions, le testeur revient sur l'étape de substitutions. Il revient alors sur le code généré, une classe Java (il précise alors de nouveau le cadre, c'est-à-dire que les applications sont construites à partir de composants Fractal implémentés avec le langage Java). Il fournit à l'utilisateur le fichier présenté dans l'annexe E. Il précise que ce fichier est généré lors de la substitution (cf. figure 5.6) de l'entrée texte provenant de l'application "Cinema" permettant de renseigner l'adresse et de l'entrée texte provenant de l'application "Maps" permettant de renseigner l'adresse de départ de l'itinéraire, l'élément graphique "gardé" étant celui provenant de l'application "Cinema".

    Sans donner d'indication supplémentaire, le testeur demande à l'utilisateur s'il arrive à comprendre le rôle de cette classe Java générée et s'il peut la compléter. Le testeur introduit ensuite un schéma de liaisons entre composants Fractal où apparaît l'adapteur généré. Le testeur demande alors de nouveau à l'utilisateur d'exprimer le rôle que joue cet adapteur lors de cette fusion.

16. Enfin, le testeur termine par un entretien basé sur le questionnaire présenté dans l'annexe G pour faire un bilan des différentes manipulations effectuées durant le tests avec l'utilisateur.

ANNEXE G

# Questionnaire de fin de test utilisateur

CETTE annexe présente le questionnaire ayant servi de base pour l'entretien de fin de test utilisateur.

## G.1 Questions générales

- Avez vous compris le processus de composition d'applications ?
- Pouvez-vous le décrire et l'expliquer ?
- Avez vous bien compris les différentes étapes de la composition d'applications ?
- De manière générale, trouvez vous ce processus utile ?

## G.2 A propos de l'étape de Sélection

- Pouvez vous décrire la procédure de cette étape de sélection et ses sous étapes ?
- Quels types d'information vous ont aidé ou auraient pu vous aider durant cette étape ?
- Avez vous compris le principe d'extensions de sélection ?
- Quelles différences voyez-vous entre les différents types d'extensions ?
- Si vous deviez refaire la sélection, utiliseriez-vous les extensions ou plutôt la sélection "simple" ?
- Quelles sont pour vous les extensions les plus utiles ?
- Comprenez vous le résultat de cette étape de sélection ? Quels éléments sont ajoutés à la sélection finale ?.

## G.3 A propos de l'étape de Substitutions

- Pouvez vous décrire la procédure de cette étape de substitutions et ses sous étapes ?
- Quels types d'information vous ont aidé ou auraient pu vous aider durant cette étape ?
- Comprenez vous le résultat de cette étape de substitutions ?
- Est-ce que le résultat obtenu correspond à vos attentes ?
- Avez-vous compris à quoi correspond les fichiers générés ?

## G.4 A propos de l'étape de Placement

- Pouvez vous décrire la procédure de cette étape de placement et ses sous étapes ?
- Quels types d'information vous ont aidé ou auraient pu vous aider durant cette étape ?

# Table des figures

# Liste des tableaux

# Liste des algorithmes

# Listings

# Résumé

## Composition d'applications multi-modèles dirigée par la composition des interfaces graphiques

Force est de constater que composer des applications existantes afin d'en réutiliser tout ou une partie est une tâche complexe. Pourtant avec l'apparition quotidienne d'applications, les éditeurs d'applications ont de plus en plus besoin d'effectuer de telles compositions pour répondre à la demande croissante des utilisateurs. Les travaux existants ne traitent généralement que d'un seul point de vue : celui du "Noyau Fonctionnel" dans le domaine du Génie Logiciel, celui des "Tâches" ou celui de l'"Interface Graphique" dans le domaine des Interactions Homme-Machine (IHM).
Cette thèse propose une nouvelle approche basée sur un modèle d'application complet (fonctionnel, tâche et interface graphique). Elle permet à un utilisateur de naviguer entre ces différents modèles pour sélectionner des ensembles cohérents pouvant être composer par substitution.
Une implémentation de cette approche a permis d'effectuer des tests utilisateurs confortant les bienfaits d'une modélisation complète.



**Mots clefs** : composition d'applications, modèle complet, interface graphique, tâches, composants logiciels, ontologies



# Abstract

## Multi-models application composition driven by user interface composition

One has to note that composing existing applications by completely or partly reusing them is a complex task. Nevertheless with the daily appearance of new available applications on any media, the application editors need to perform such compositions more and more to answer the increasing users' requests. Modeling an application for composition or just determining by which point of view on applications make this composition is not easy. Works exist, but generally deal or ensue from only a single point of view : the "Functional Core" point of view in Software Engineering field, the "Task" one or "User Interface" one in Human Computer Interaction (HCI) field.
This thesis defines a new approach based on a complete application model (functionnal, task and user interface). It enables an user to navigate between those different models in order to select consistent sets. These last ones are composable by substitution.
An implementation of this approach was used to perform user tests whose results consolidate benefits of a complete model.



**Keywords** : application composition, complete model, user interface, tasks, software components, ontologies